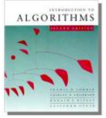
Big O Notation

- $O(g(n)) = \{ f(n) \mid \exists\, c > 0$, and n_0 , so that $0 \le f(n) \le cg(n)$ for all $n \ge n_0 \}$

- $f(n) = O(g(n))$ means $f(n) \in O(g(n))$ (i.e, at most)

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \le \mathrm{c}$$



Omega Ω Notation

- $\Omega(g(n)) = \{ f(n) \mid \exists c > 0$, and n_0, so that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$

- $f(n) = \Omega(g(n))$ means f(n) $\in \Omega(g(n))$ (i.e, at least)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq \text{c}$$

Theta Θ Notation

- Combine lower and upper bound
  - $f(n)=\Theta(g(n))$ means $f(n)=O(g(n))$ and $f(n) = \Omega(g(n))$
- Means tight: of the same order
- $\Theta(g(n)) = \{ f(n) \mid \exists\, a,b > 0$, and n_0 , so that $0 \leq ag(n) \leq f(n) \leq bg(n)$ for all $n \geq n_0\}$
  - $f(n)=\Theta(g(n))$ means $g(n)=\Theta(f(n))$?

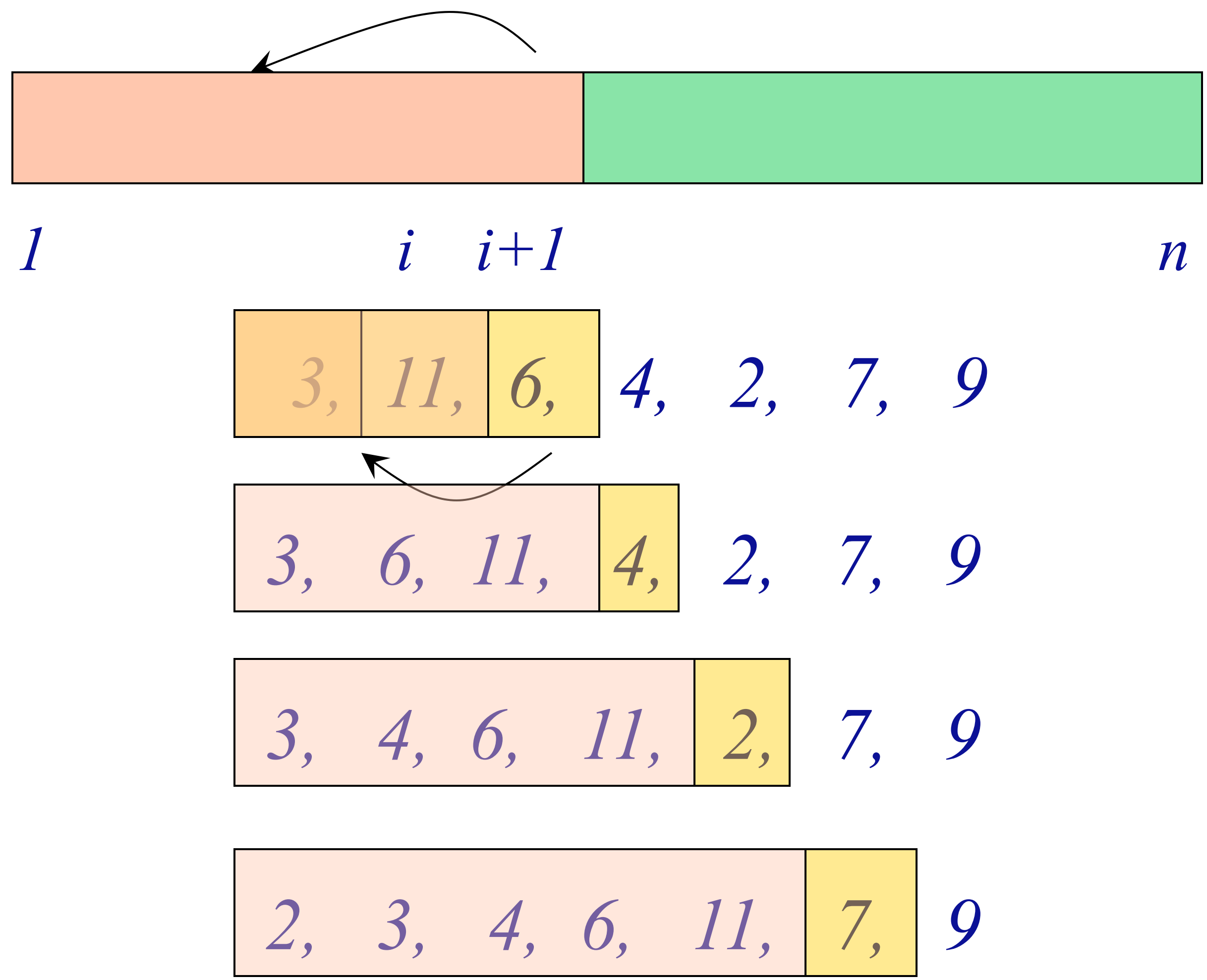
- Insertion sort:
  - Worst case running time is $\Theta(n^2)$.

Sorting problem

- Input: $A = \langle a_1, a_2, \ldots a_n \rangle$
- Output: permutation of A that is sorted

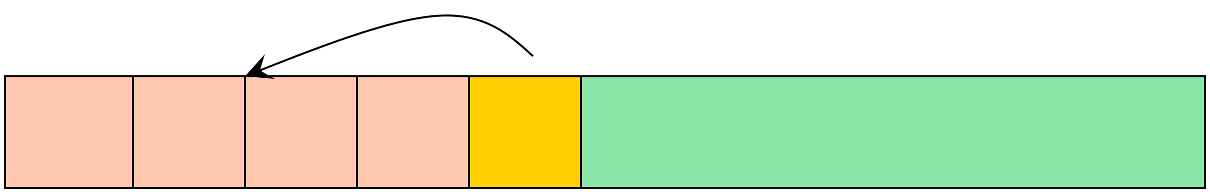
- Example:
  - Input: $\langle 3, 11, 6, 4, 2, 7, 9 \rangle$
  - Output: $\langle 2, 3, 4, 6, 7, 9, 11 \rangle$

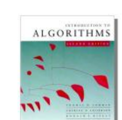
Insertion Sort



Pseudo-code

```
InsertionSort(A, n)
  for i = 2 to n do
      key = A[i]
      j = i - 1
      while j > 0 and A[j] > key do
            A[j+1] = A[j]
            j = j - 1
      A[j+1] = key
```





Insertion Sort

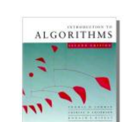
```
InsertionSort(A, n)
    for  i = 2  to  n   do
        key = A[i]                  ——————  c_1
        j = i - 1
        while  j > 0  and  A[j] > key   do
            A[j+1] = A[j]           ——————  c_2
            j = j - 1
        A[j+1] = key
```

$$T(n) \leq \sum_{i=2}^{n} (c_1 + c_2 i) = c_3 n^2 + c_4 n + c_5$$

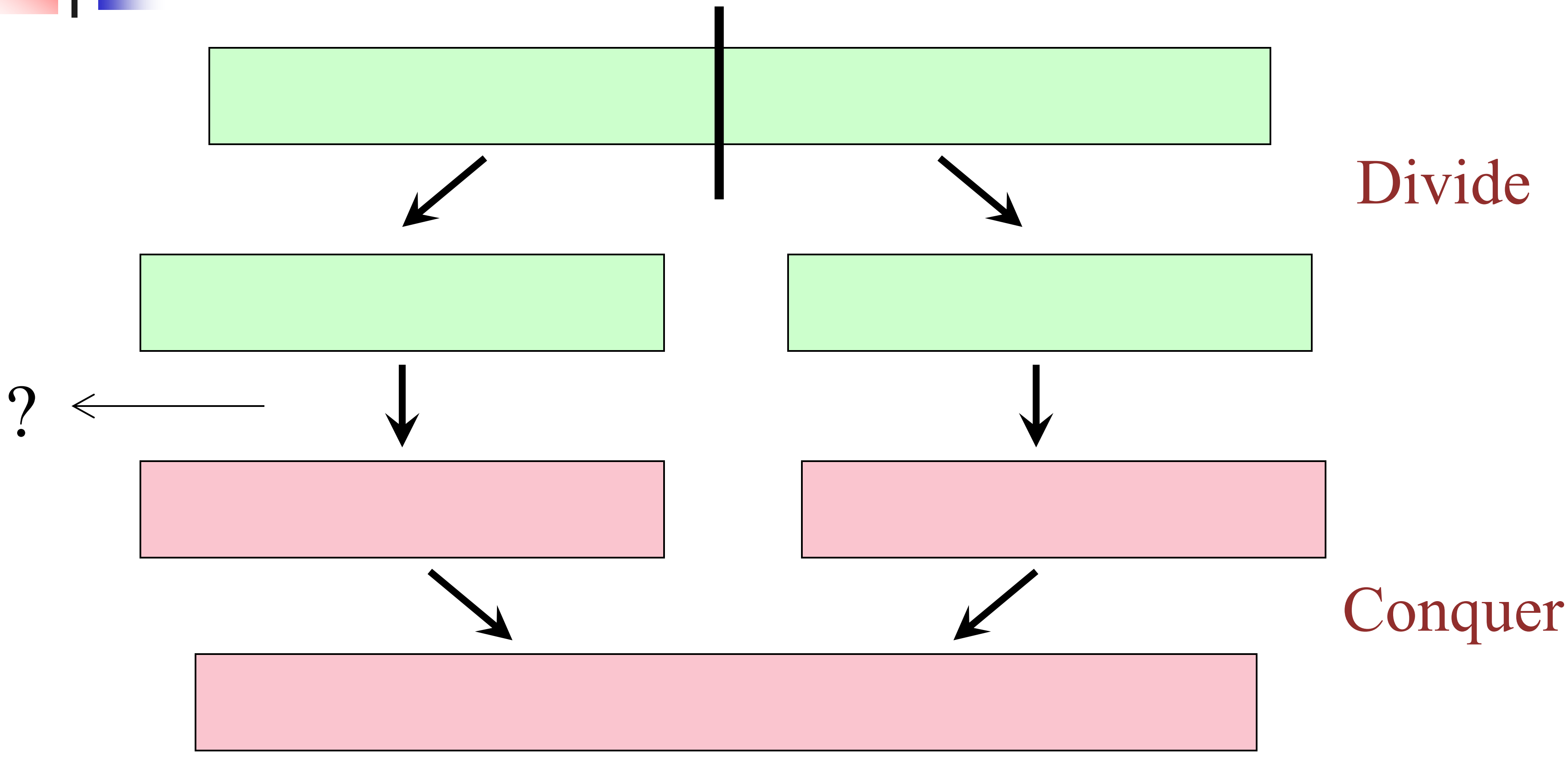
Sorting Revisited

- Insertion sort:
  - Worst case: $\Theta(n^2)$
- Can we do better?

Yes !
Merge sort
Use a Divide-and-Conquer Paradigm

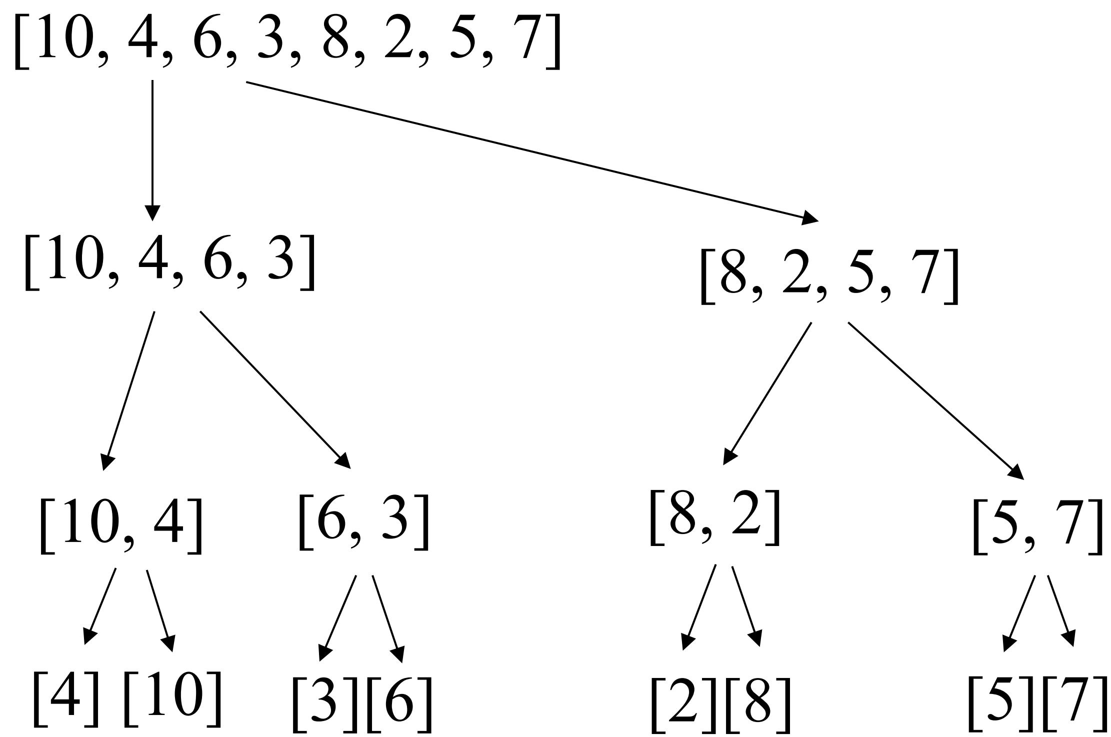


Merge Sort



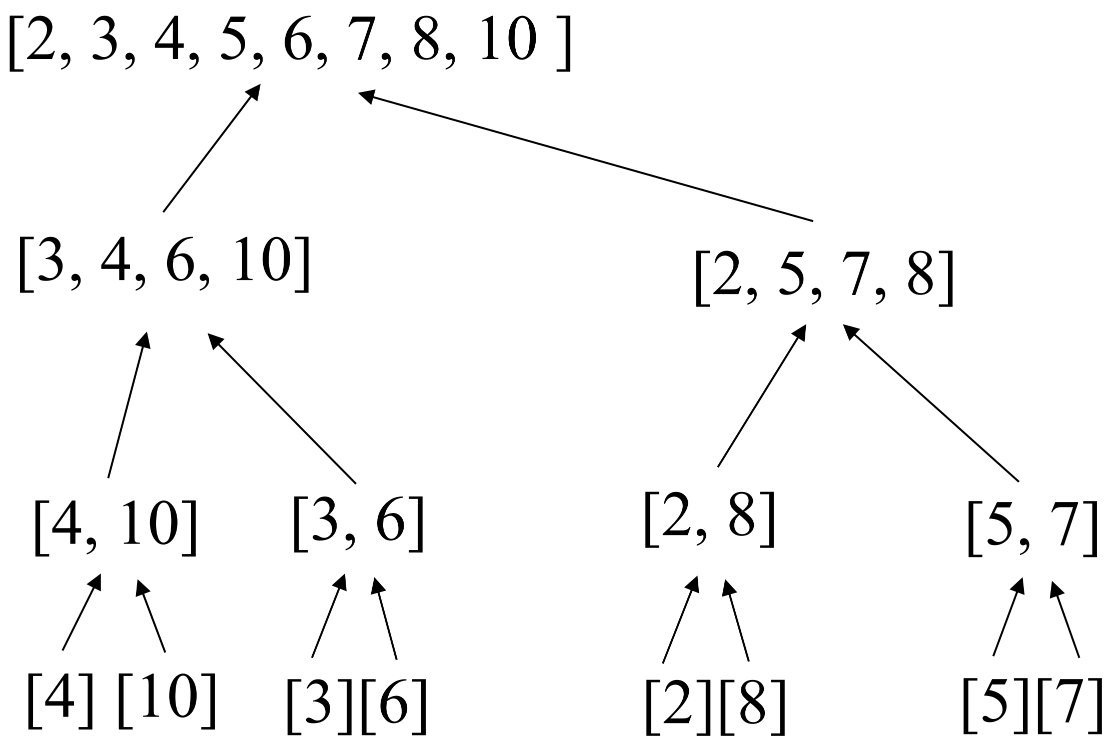
Example

- Partition into lists of size n/2



Example Cont'd

- Merge



Merge Sort

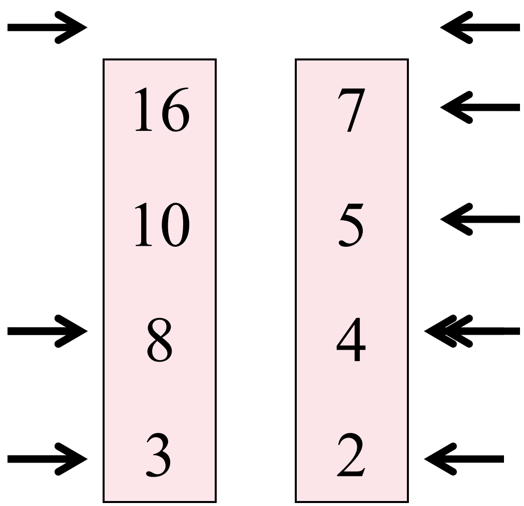
```
MergeSort(A, left, right) {
   if (left < right) {
        mid = floor((left + right) / 2);
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);
        Merge(A, left, mid, right);
   }
}

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A
//    (how long should this take?)
```

Analysis of Merge Sort

Statement

```
MergeSort(A, left, right) {
 if (left < right) {
    mid = floor((left + right) / 2);
    MergeSort(A, left, mid);
    MergeSort(A, mid+1, right);
    Merge(A, left, mid, right);
 }
}
```

To Merge Sorted (B, C)



2 2 3 4 5 7 8 10 16

If size of *B* and *C* are *s* and t:

Running time: $\boxed{\Theta(s + t)}$

Pseudocode

MergeSort (*A, 1, n*)

$m = n / 2$;

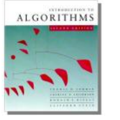
$A1$ = MergeSort (*A, 1, m*);

$A2$ = MergeSort (*A, m+1, n*);

Merge (*A1, A2*);

Worst case time complexity:

$$T(n) = 2T(n/2) + O(n)$$



Recurrences

- Solution for *$T(n) = 2T(n/2) + n$*

- $T(n) = 2\,T(n/2) + n$

  $= 2\,(2\,T\,(n/4) + n/2) + n \;= 4\,T(n/4) + n + n$

  $= 4\,(2\,T(n/8) + n/8)) + n + n$

  ....

  $= \Theta(n \lg n)$

Recursion-tree Method

- Solve $T(n) = 2\,T(n/2) + n$

$T(n) =$

n

$T(n/2)$ $\quad$ $T(n/2)$

$T(n/4)$ $\quad$ $T(n/4)$ $\quad$ $T(n/4)$ $\quad$ $T(n/4)$

$\Theta(1)$ $\cdots$ $\Theta(1)$

n

$n/2 + n/2$

$4 \times (n/4)$

$\vdots$

$n \times \Theta(1)$

$$T(n) \le n \times \text{height(tree)} = n \lg n$$

Changing n to cn does not change asymptotic complexity.

Other Examples

MergeSort ($A, 1, n$)

$m = n / 3$;

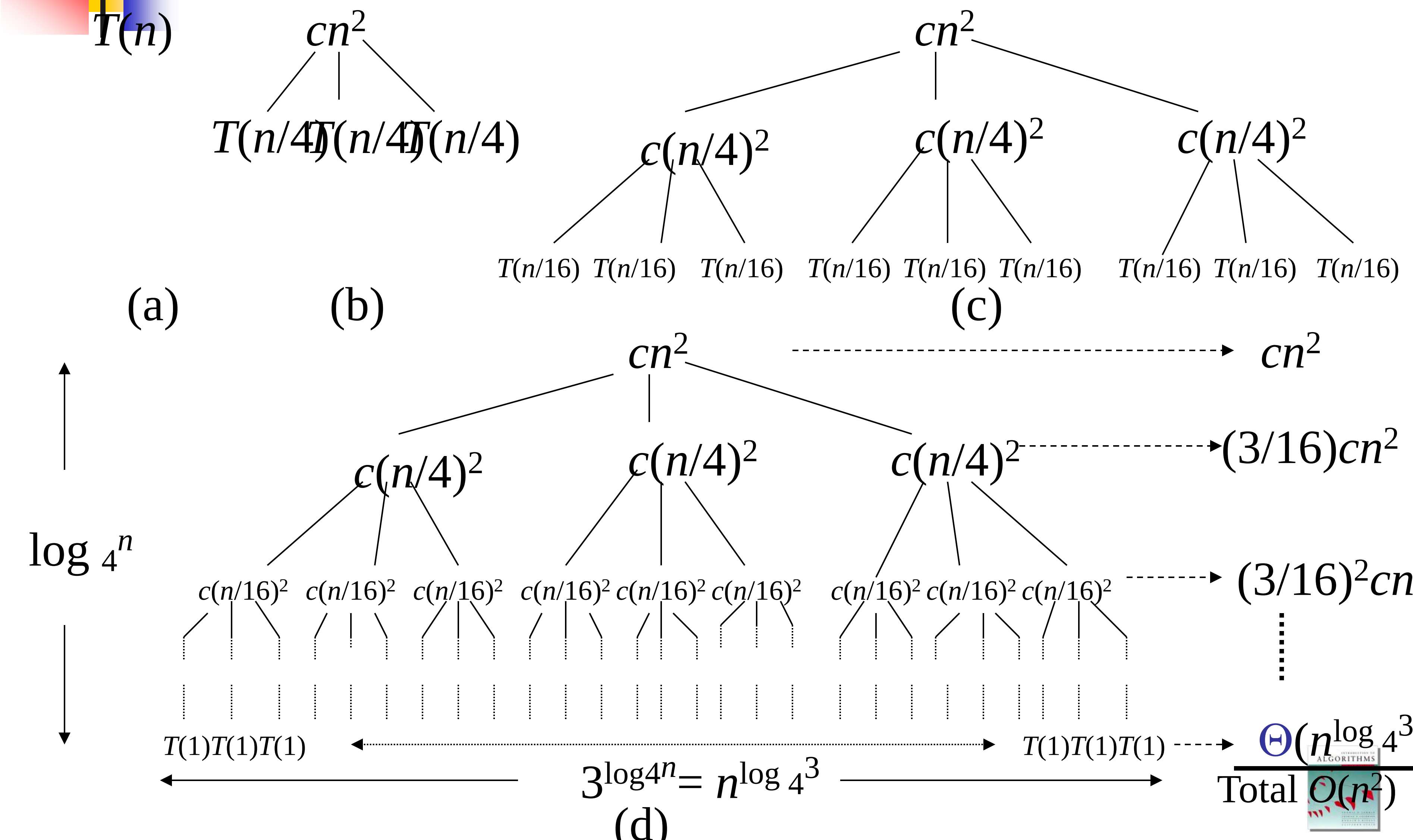
$A1$ = MergeSort ($A, 1, m$);

3-way merge sort: $T(n) = 3T(n/3) + n$

Merge ($A1, A2$);

$$T(n) = T(n/3) + T(2n/3) + n$$

Recursion Tree for $T(n)=3T(\lfloor n/4 \rfloor)+\Theta(n^2)$



Solution to $T(n)=3T(\lfloor n/4 \rfloor)+\Theta(n^2)$

- The height is $\log_4 n$,
- #leaf nodes $= 3^{\log_4 n} = n^{\log_4 3}$. Leaf node cost: $T(1)$.
- Total cost $T(n)=cn^2+(3/16)\,cn^2+(3/16)^2\,cn^2+ \cdots +(3/16)^{\log_4 n-1}\,cn^2+\Theta(n^{\log_4 3})$
  $=(1+3/16+(3/16)^2+ \cdots +(3/16)^{\log_4 n-1})\,cn^2+\Theta(n^{\log_4 3})$
  $<(1+3/16+(3/16)^2+ \cdots +(3/16)^m+ \cdots)\,cn^2+\Theta(n^{\log_4 3})$
  $=(1/(1-3/16))\,cn^2+\Theta(n^{\log_4 3})$
  $=16/13cn^2+\Theta(n^{\log_4 3})$
  $=O(n^2)$.

Prove the above Guess

- $T(n)=3T(\lfloor n/4 \rfloor)+\Theta(n^2) = O(n^2)$.
- Show $T(n) \le dn^2$ for some d.
- $T(n) \le 3(d\,(\lfloor n/4 \rfloor)^2) + cn^2$

$$\le 3(d\,(n/4)^2) + cn^2$$

$$= 3/16(dn^2) + cn^2$$

$$\le dn^2, \quad \text{as long as } d \ge (16/3)c.$$

One more example

- $T(n)=T(n/3)+ T(2n/3)+O(n)$.
- Construct its recursive tree
- $T(n)=O(cn\lg_{3/2}{}^{n}) = O(n\lg n)$.
- Prove $T(n) \leq dn\lg n$.

Recursion Tree of $T(n)=T(n/3)+T(2n/3)+O(n)$

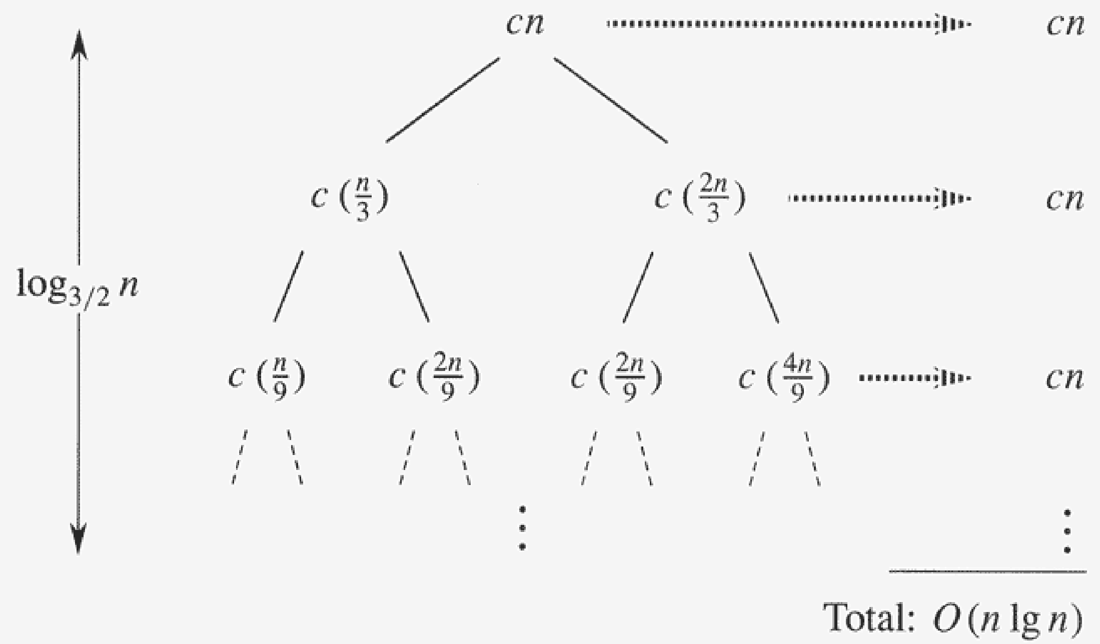
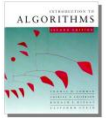


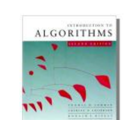
Figure 4.2 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.



Solving Recurrences

- The "iteration method"
  - Expand the recurrence
  - Work some algebra to express as a summation
  - Evaluate the summation
- We showed several examples, were in the middle of:

$$\boxed{T(n) = \begin{cases} c & n = 1 \\ aT\left(\dfrac{n}{b}\right) + cn & n > 1 \end{cases}}$$



$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\dfrac{n}{b}\right) + cn & n > 1 \end{cases}$$

- $T(n) =$

  $aT(n/b) + cn$

  $a(aT(n/b/b) + cn/b) + cn$

  $a^2T(n/b^2) + cna/b + cn$

  $a^2T(n/b^2) + cn(a/b + 1)$

  $a^2(aT(n/b^2/b) + cn/b^2) + cn(a/b + 1)$

  $a^3T(n/b^3) + cn(a^2/b^2) + cn(a/b + 1)$

  $a^3T(n/b^3) + cn(a^2/b^2 + a/b + 1)$

  …

  $a^kT(n/b^k) + cn(a^{k-1}/b^{k-1} + a^{k-2}/b^{k-2} + \ldots + a^2/b^2 + a/b + 1)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So we have
  - $T(n) = a^k T(n/b^k) + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$
- For $k = \log_b n$
  - $n = b^k$
  - $T(n) = a^k T(1) + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$

    $= a^k c + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$

    $= ca^k + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$

    $= cna^k /b^k + cn(a^{k-1}/b^{k-1} + \ldots + a^2/b^2 + a/b + 1)$

    $= cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\dfrac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a = b$?
  - $T(n) = cn(k + 1)$
    $= cn(\log_b n + 1)$
    $= \Theta(n \log n)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a < b$?

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a < b$?
  - Recall that $\Sigma\ (x^k + x^{k-1} + \ldots + x + 1) = (x^{k+1} - 1)/(x-1)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a < b$?
  - Recall that $(x^k + x^{k-1} + \ldots + x + 1) = (x^{k+1} - 1)/(x-1)$
  - So:

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 = \frac{(a/b)^{k+1} - 1}{(a/b) - 1} = \frac{1 - (a/b)^{k+1}}{1 - (a/b)} < \frac{1}{1 - a/b}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a < b$?
  - Recall that $\Sigma(x^k + x^{k-1} + \ldots + x + 1) = (x^{k+1} - 1)/(x-1)$
  - So:

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 = \frac{(a/b)^{k+1} - 1}{(a/b) - 1} = \frac{1 - (a/b)^{k+1}}{1 - (a/b)} < \frac{1}{1 - a/b}$$

$$T(n) = cn \cdot \Theta(1) = \Theta(n)$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + ... + a^2/b^2 + a/b + 1)$
- What if $a > b$?

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\dfrac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a > b$?

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 \quad = \quad \frac{(a/b)^{k+1} - 1}{(a/b) - 1} \quad = \quad \Theta\left((a/b)^k\right)$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a > b$?

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 \quad = \quad \frac{(a/b)^{k+1} - 1}{(a/b) - 1} \quad = \quad \Theta\left((a/b)^k\right)$$

  - $T(n) = cn \cdot \Theta(a^k / b^k)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a > b$?

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 \quad = \quad \frac{(a/b)^{k+1} - 1}{(a/b) - 1} \quad = \quad \Theta\left((a/b)^k\right)$$

  - $T(n) = cn \cdot \Theta(a^k / b^k)$

$$= cn \cdot \Theta(a^{\log n} / b^{\log n}) = cn \cdot \Theta(a^{\log n} / n)$$

$$T(n)=\begin{cases} c & n=1 \\ aT\left(\frac{n}{b}\right)+cn & n>1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a > b$?

$$\frac{a^k}{b^k}+\frac{a^{k-1}}{b^{k-1}}+\cdots+\frac{a}{b}+1 \quad = \quad \frac{(a/b)^{k+1}-1}{(a/b)-1} \quad = \quad \Theta\left((a/b)^k\right)$$

  - $T(n) = cn \cdot \Theta(a^k / b^k)$

$$= cn \cdot \Theta(a^{\log n} / b^{\log n}) = cn \cdot \Theta(a^{\log n} / n)$$

recall logarithm fact: $a^{\log n} = n^{\log a}$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a > b$?

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 \quad = \quad \frac{(a/b)^{k+1} - 1}{(a/b) - 1} \quad = \quad \Theta\left((a/b)^k\right)$$

  - $T(n) = cn \cdot \Theta(a^k / b^k)$

$= cn \cdot \Theta(a^{\log n} / b^{\log n}) = cn \cdot \Theta(a^{\log n} / n)$

recall logarithm fact: $a^{\log n} = n^{\log a}$

$= cn \cdot \Theta(n^{\log a} / n) = \Theta(cn \cdot n^{\log a} / n)$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So with $k = \log_b n$
  - $T(n) = cn(a^k/b^k + \ldots + a^2/b^2 + a/b + 1)$
- What if $a > b$?

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 \quad = \quad \frac{(a/b)^{k+1} - 1}{(a/b) - 1} \quad = \quad \Theta\left((a/b)^k\right)$$

  - $T(n) = cn \cdot \Theta(a^k / b^k)$

$= cn \cdot \Theta(a^{\log n} / b^{\log n}) = cn \cdot \Theta(a^{\log n} / n)$

recall logarithm fact: $a^{\log n} = n^{\log a}$

$= cn \cdot \Theta(n^{\log a} / n) = \Theta(cn \cdot n^{\log a} / n)$

$= \Theta(n^{\log a})$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\dfrac{n}{b}\right) + cn & n > 1 \end{cases}$$

- So…

$$T(n) = \begin{cases} \Theta(n) & a < b \\ \Theta(n \log_b n) & a = b \\ \Theta\left(n^{\log_b a}\right) & a > b \end{cases}$$

Substitution Method

- Solution for $T(n) = 2T(n/2) + n$

Guess a solution,
Verify its correctness

Substitution Method

- Guess $T(n) = O(n \lg n)$ i.e, $T(n) \leq c\, n \lg n$
- Induction:
  - Assume true for T(n/2)
  - T(n) $= 2\,T(n/2) + n$

    $\leq 2\,c\,n/2 \lg(n/2) + n$

    $= c\,n \lg n - (c-1)\,n$

    $\leq c\,n \lg n$ if c > 1
  - Base case:
    - $T(1)$ …. no
    - But true for $T(2)$ for sufficiently large c

Verification

Solve c

More Examples

- $T(n) = 2T(n/2) + 5$
  - $T(n) = \Theta(n)$
- $T(n) = T(n/10) + T(9n/10) + n$

Use expansion / recursion tree when the input recurrence is clean. When constants/coefficients/factors are no longer nice, use expansion / recursion tree to obtain a guess, or make an educated guess through other ways, and verify the guess using the substitution method.

The Master Theorem

- Given: a *divide and conquer* algorithm
  - An algorithm that divides the problem of size *n* into *a* subproblems, each of size *n*/*b*
  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function *f*(n)
- Then, the Master Theorem gives us a cookbook for the algorithm’s running time:

The Master Theorem

- if T(n) = aT(n/b) + f(n) then

$$T(n) = \left\{ \begin{array}{ll} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{array} \right\} \begin{array}{l} \varepsilon > 0 \\ c < 1 \end{array}$$

The Master Method

- The Master method applies to recurrences of the form

  $$T(n) = a\,T(n/b) + f(n)$$

  where $a \geq 1$ and $b > 1$, and $f(n)$ is an asymptotically positive function.

Case One

- $f(n) = O(n^{\log_b a - \varepsilon})$ for some const. $\varepsilon > 0$

  then , $T(n) = \Theta(n^{\log_b a})$

  - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by n^{ε})
  - the summation of $f(n)$ from each levels in recursion tree is consumed by n^{ε}

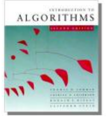
Case Two

- $f(n) = \Theta(n^{\log_b a})$ for some const. $\varepsilon > 0$
  then , $T(n) = \Theta(n^{\log_b a} \lg n)$

Case Three

- $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some const. $\varepsilon > 0$ and if $af(n/b) \leq cf(n)$ for all sufficiently large n, then, $T(n) = \Theta(f(n))$.

Note that the three cases are not complete.
There are gaps among them.



Using The Master Method

- $T(n) = 9T(n/3) + n$
  - $a=9$, $b=3$, $f(n) = n$
  - $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$
  - Since $f(n) = O(n^{\log_3 9 - \varepsilon})$, where $\varepsilon=1$, case 1 applies:

$$T(n) = \Theta\left(n^{\log_b a}\right) \text{ when } f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$

  - Thus the solution is $T(n) = \Theta(n^2)$

Application of Master Theorem

- $T(n) = T(2n/3)+1$
  - a=1,b=3/2, $f(n)$ =1
  - $n^{\log_b a} = n^{\log_{3/2} 1} = \Theta\ (n^0) = \Theta\ (1)$
  - By case 2, $T(n) = \Theta(\lg n)$.

Application of Master Theorem

- $T(n) = 3T(n/4)+n\lg n$;
  - a=3,b=4, $f(n) = n\lg n$
  - $n^{\log_b a} = n^{\log_4 3} = \Theta\ (n^{0.793})$
  - $f(n) = \Omega(n^{\log_4 3+\varepsilon})$ for $\varepsilon \approx 0.2$
  - Moreover, for large n, the "regularity" holds for c=3/4.
    - $af(n/b) = 3(n/4)\lg (n/4) \le (3/4)n\lg n = cf(n)$
  - By case 3, $T(n) = \Theta\ (f(n)) = \Theta\ (n\lg n)$.

Examples

- T(n) = 9T(n/3) + n
  - $a = 9, b = 3, f(n) = n = O(n^{\log_3 9 - \varepsilon})$
  - Case one: $T(n) = \Theta(n^2)$
- T(n) = T(n/2) + 1
  - $a = 1, b = 2, f(n) = 1 = \Theta(n^{\log_2 1}) = \Theta(1)$
  - Case two: $T(n) = \Theta(\lg n)$
- T(n) = 2T(n/2) + nlg n
  - $a = 2, b = 2, f(n) = n \lg n$
  - $n\lg n = \Omega(n^{1+\varepsilon})$ for any const. $\varepsilon > 0$

Divide and Conquer

QuickSort (*A, r, s*)

If $(r \geq s)$ return;

m = Partition (*A, r, s*);

A1 = QuickSort (*A, r, m-1*);

A2 = QuickSort (*A, m+1, s*);

~~Merge (*A1, A2*);~~

MergeSort (*A, 1, n*)

if (n < 2) return;

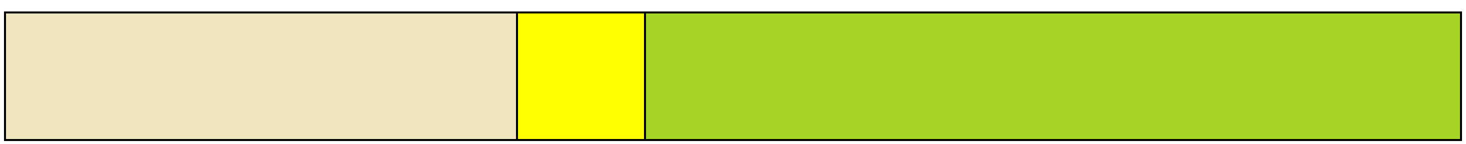
m = *n* / 2;

A1 = MergeSort (*A, 1, m*);

A2 = MergeSort (*A, m+1, n*);

Merge (*A1, A2*);

In-place sorting.



A[m]: pivot

Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.

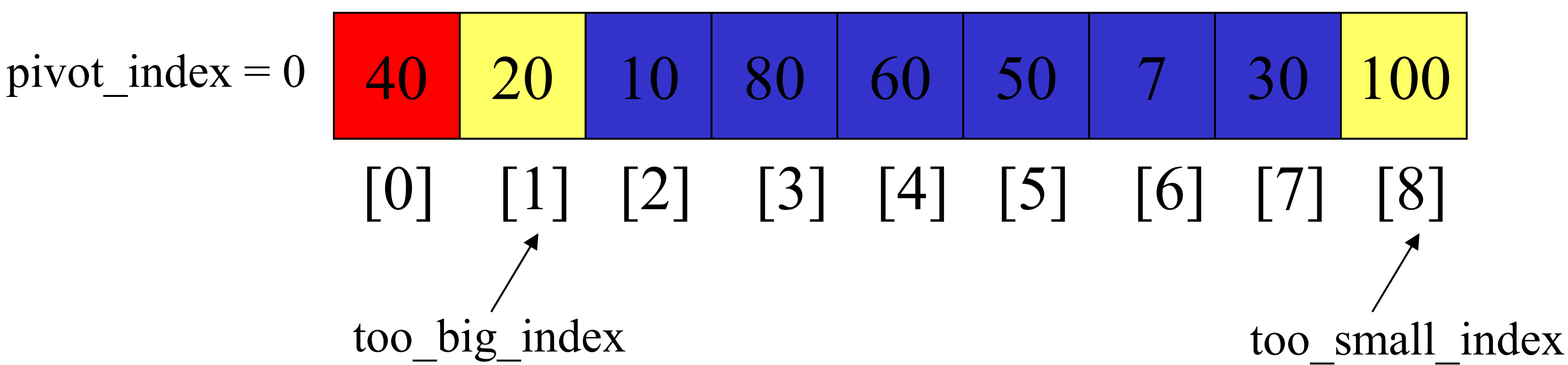
pivot_index = 0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

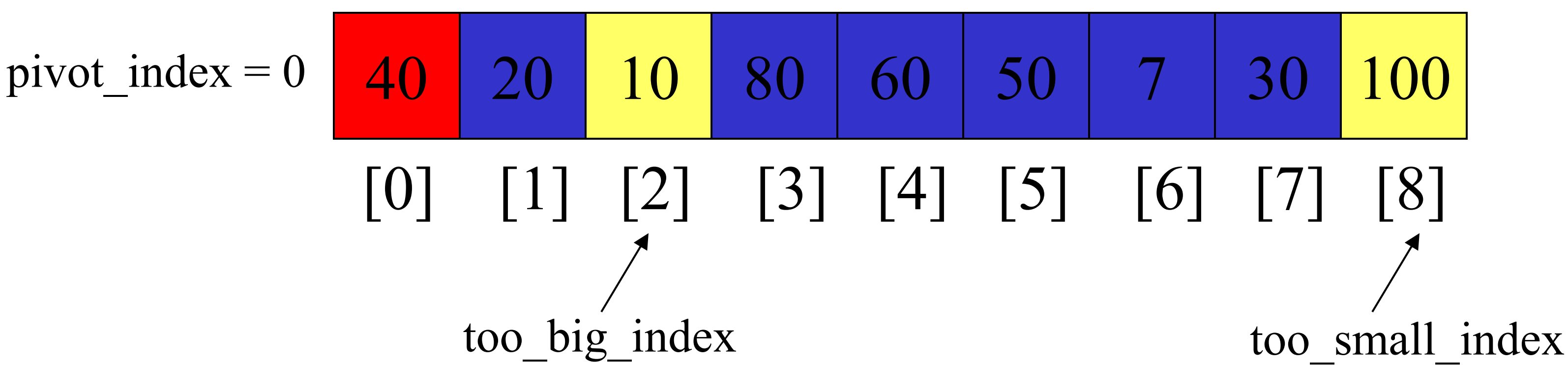
too_big_index

too_small_index

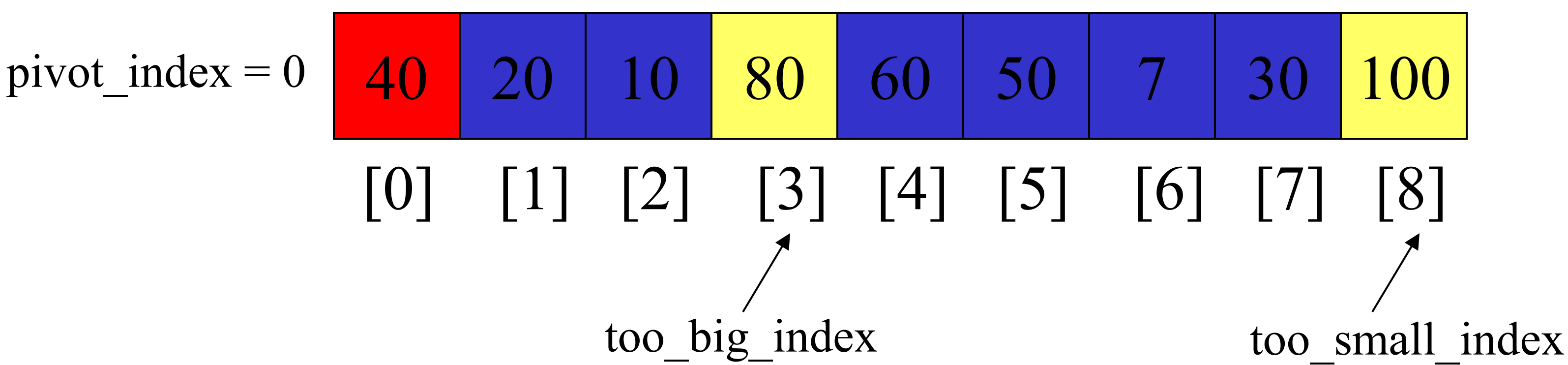
1. While data[too_big_index] <= data[pivot]
   ++too_big_index



1. While data[too_big_index] <= data[pivot]
   ++too_big_index



1. While data[too_big_index] <= data[pivot]
    ++too_big_index



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index

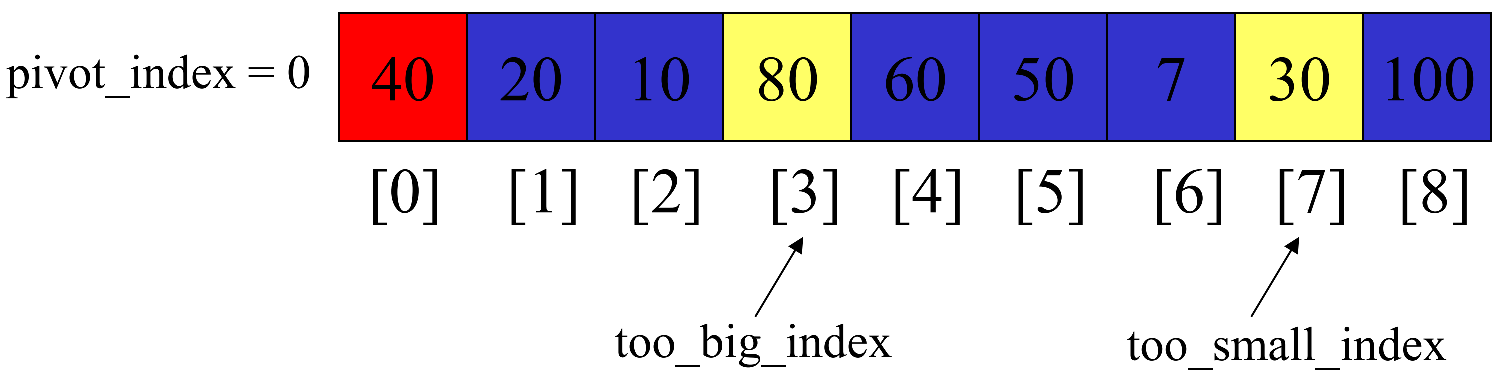
pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	80	60	50	7	30	100

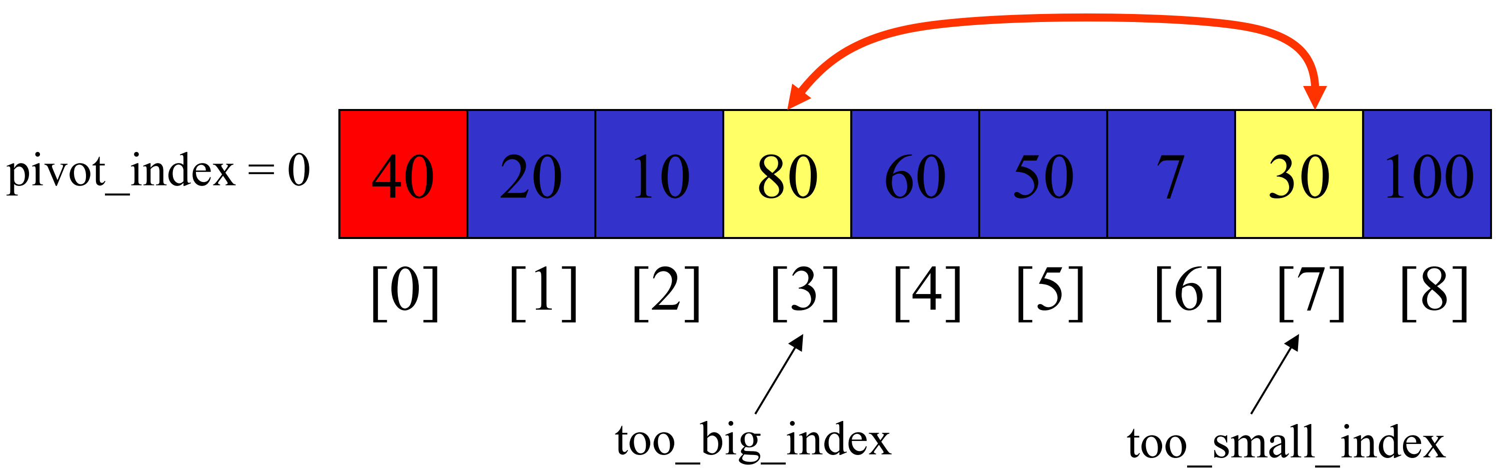
too_big_index → [3]

too_small_index → [8]

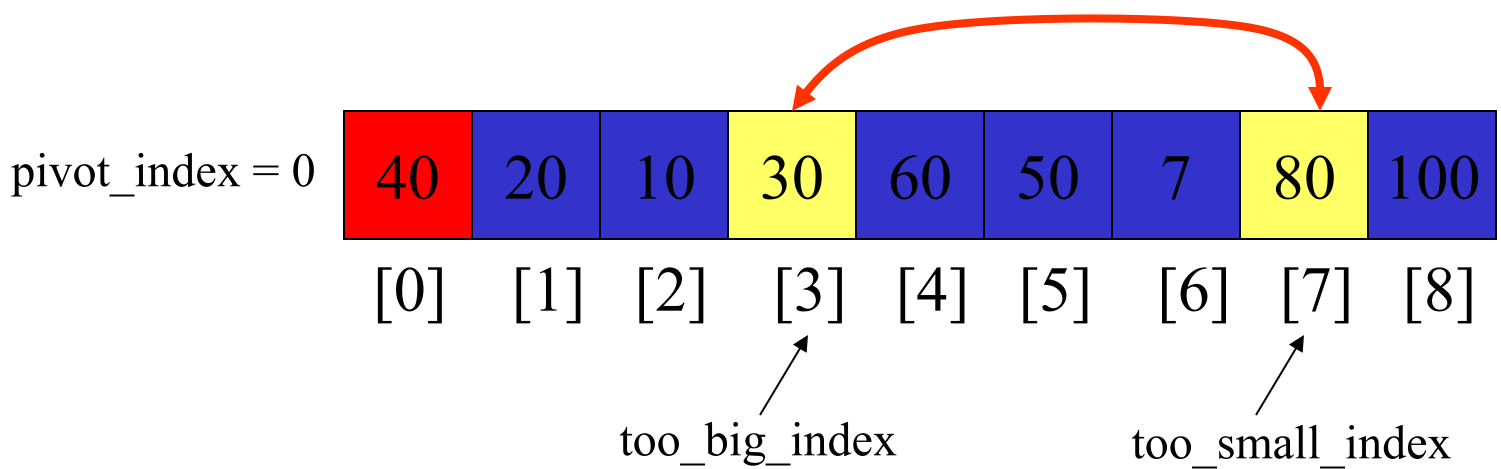
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



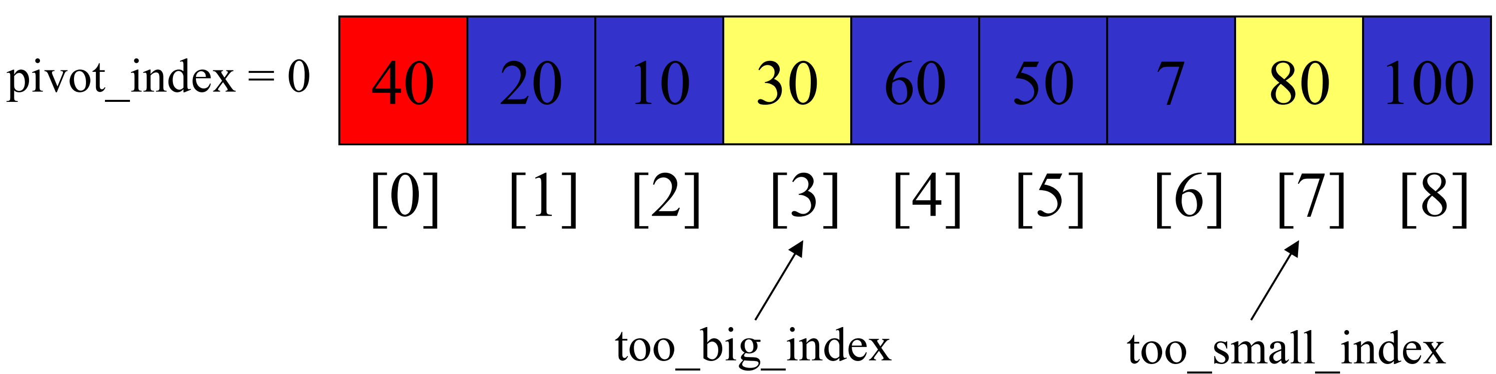
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



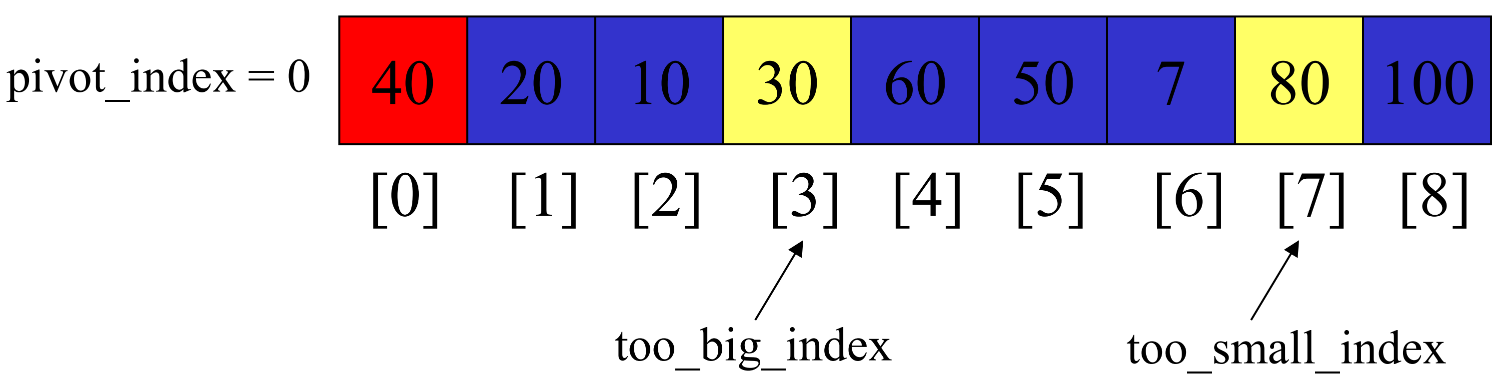
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



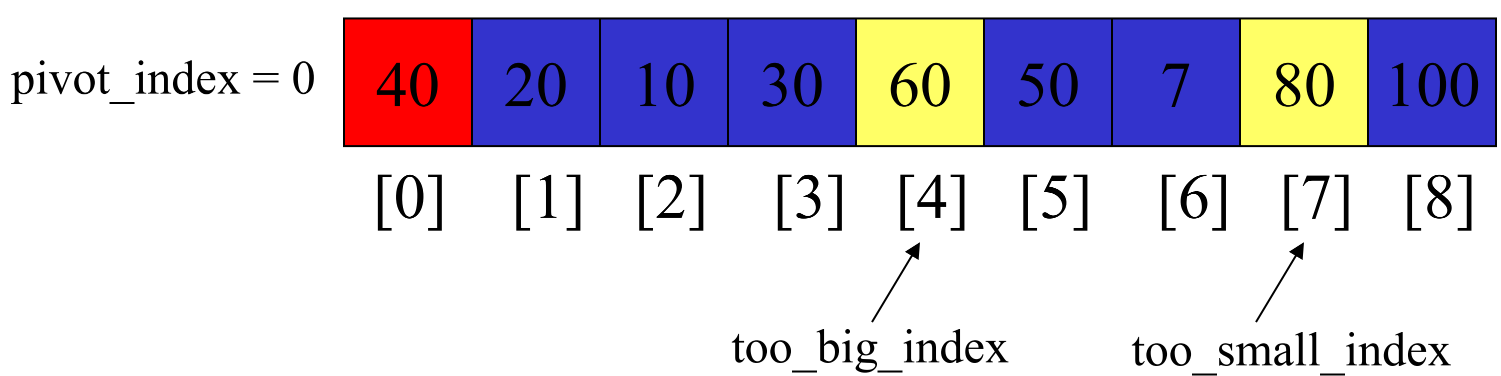
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

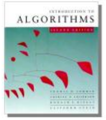


1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

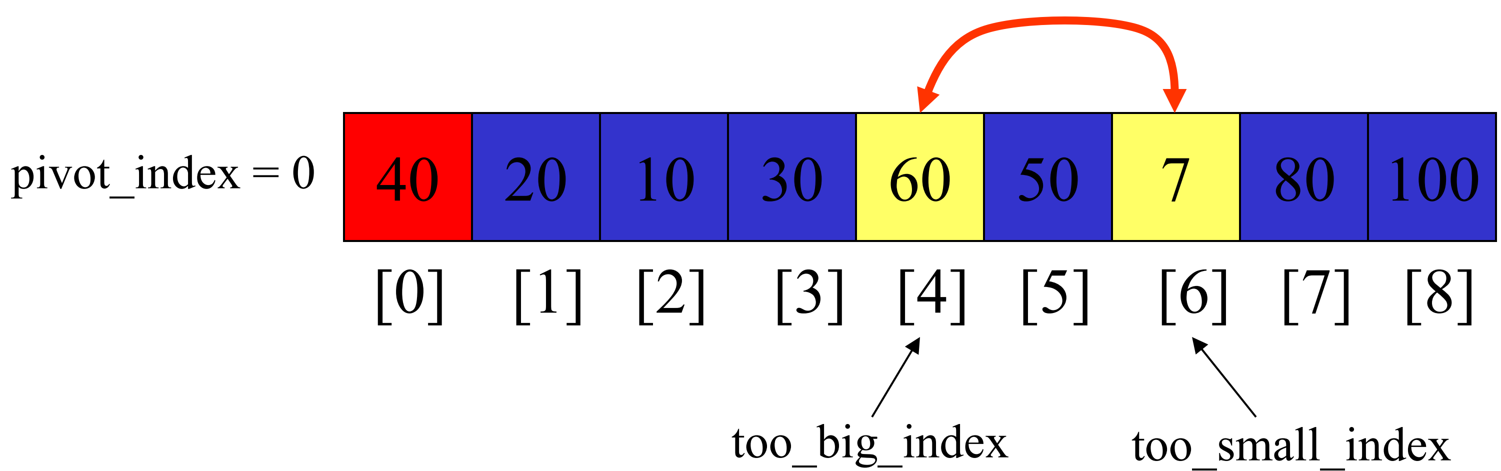
pivot_index = 0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

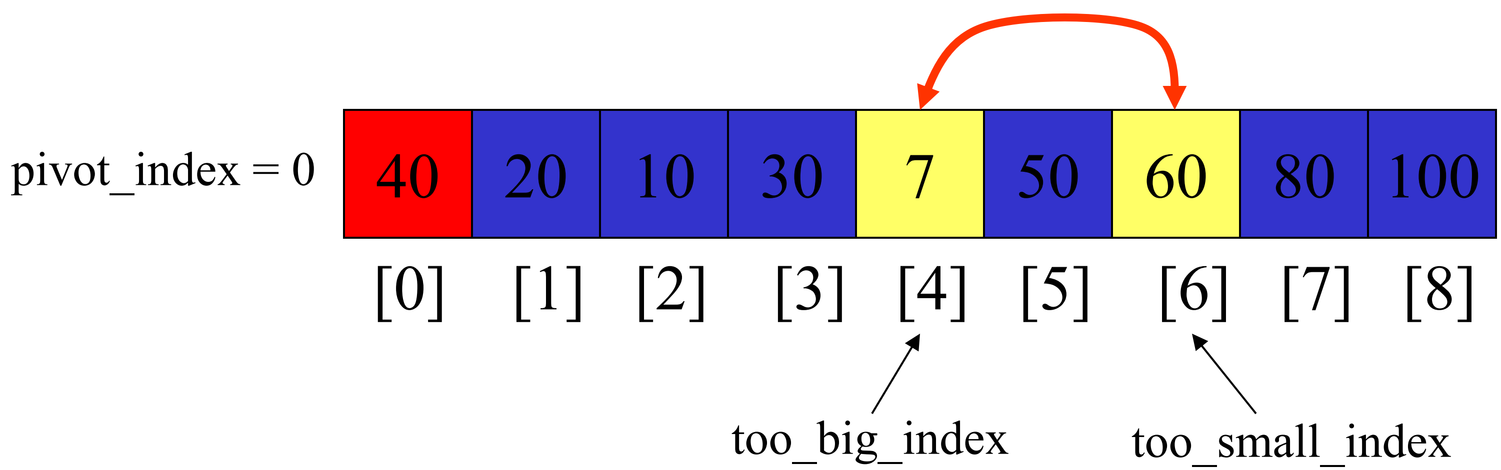
too_big_index   too_small_index



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

	40	20	10	30	7	50	60	80	100
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

too_big_index → [4]

too_small_index → [6]

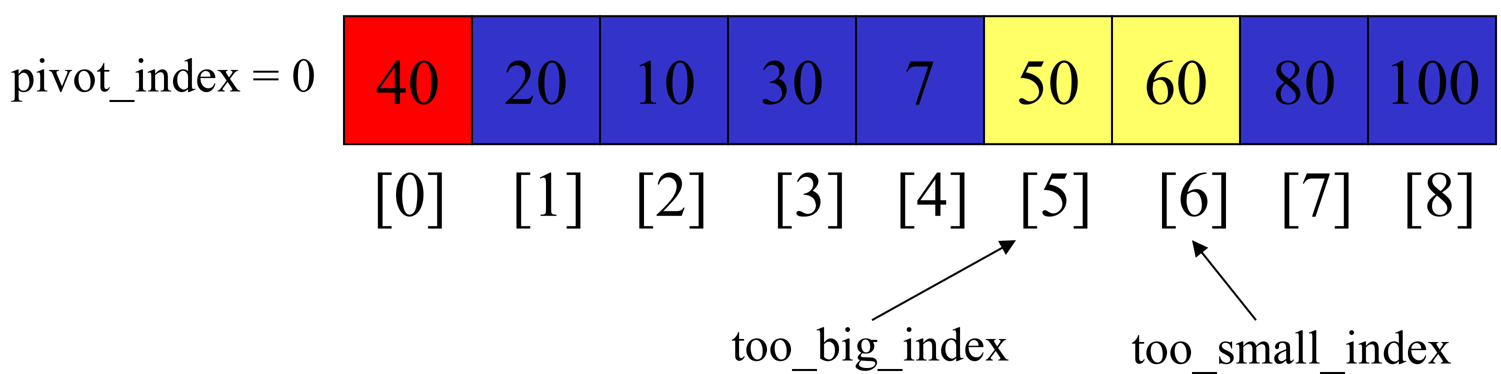
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

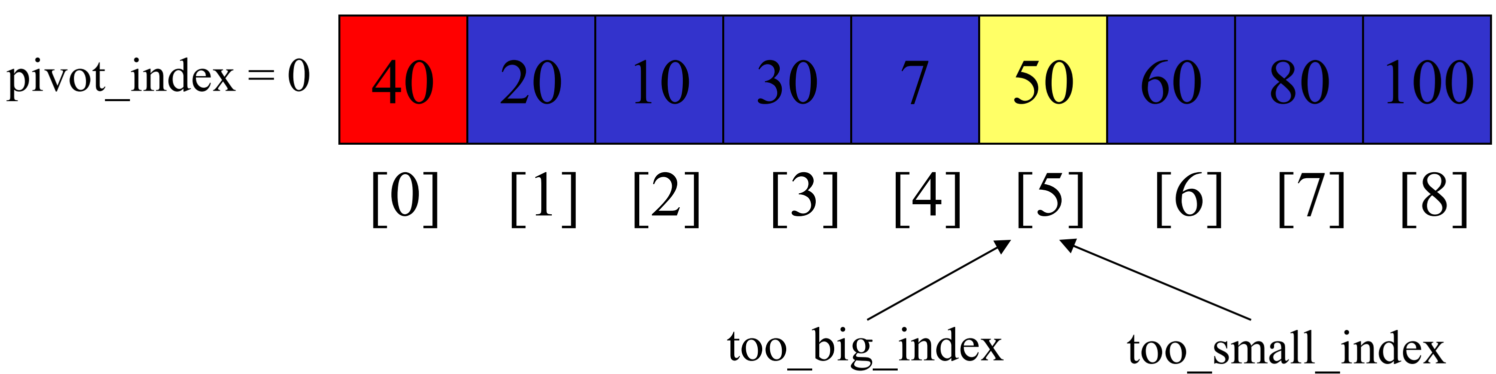
40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

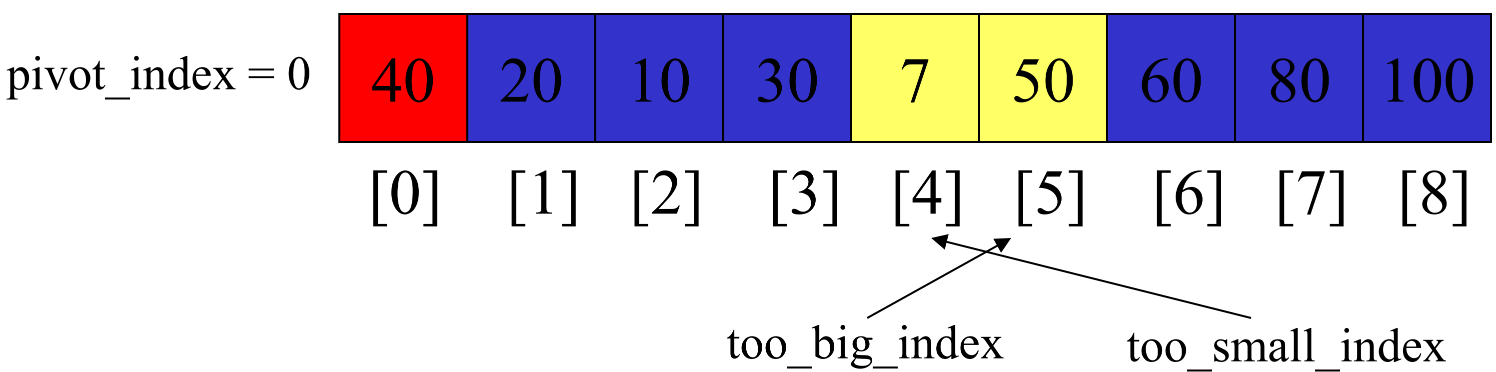
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

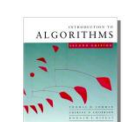


1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.





1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index
3. If too_big_index < too_small_index
    swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

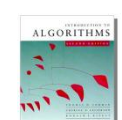
too_big_index   too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.

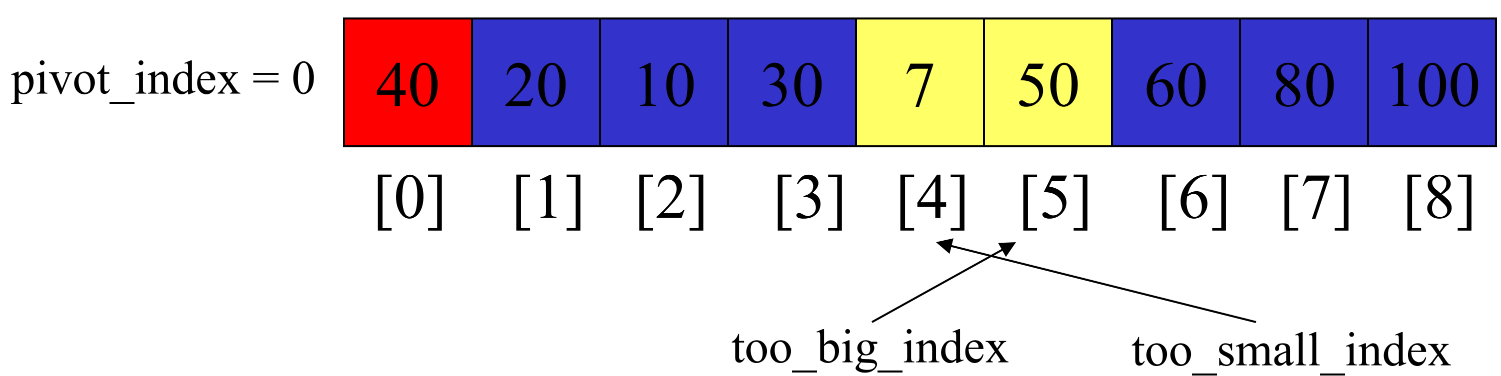
pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

too_big_index   too_small_index



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



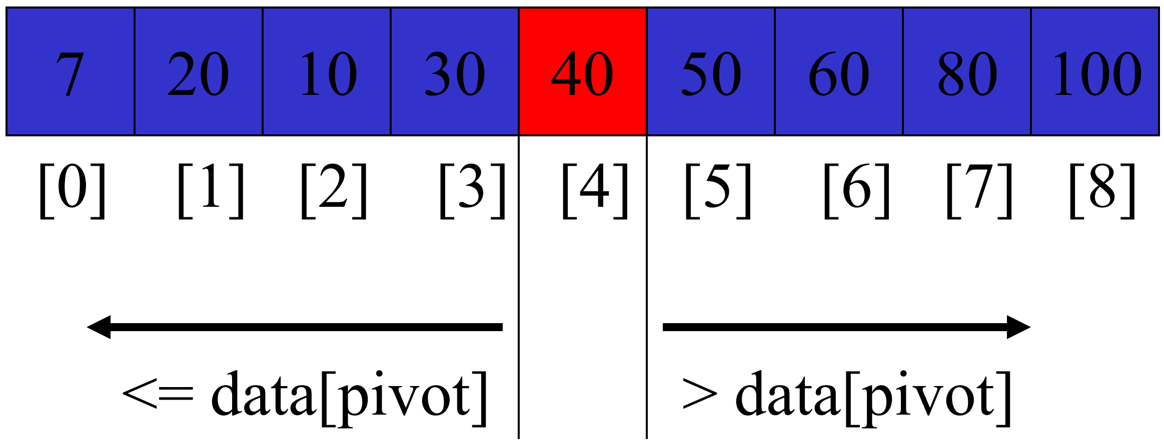
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

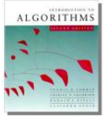
pivot_index = 4

7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

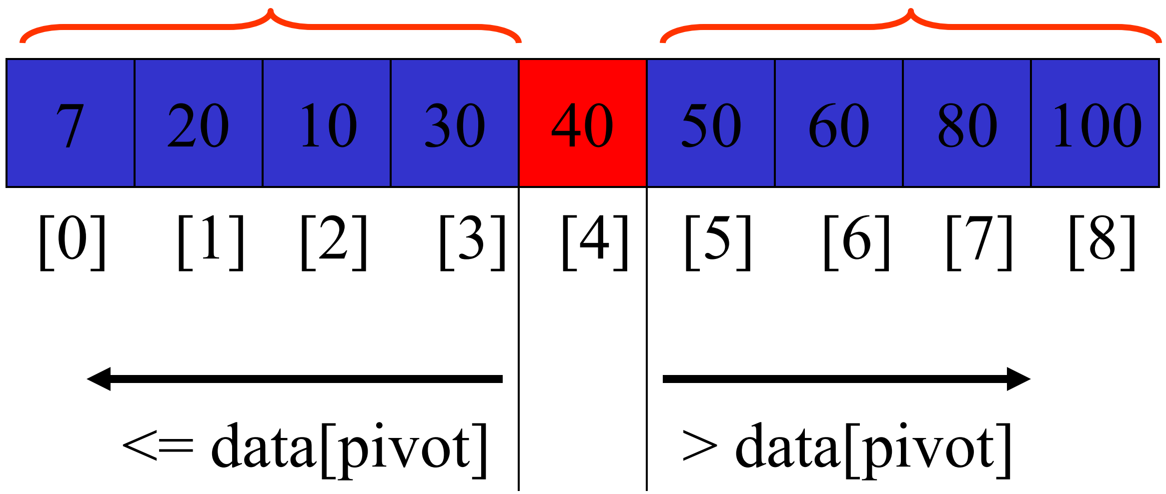
too_big_index too_small_index

Partition Result





Recursion: Quicksort Sub-arrays



Quicksort Analysis

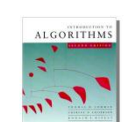
- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition?

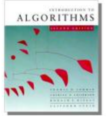
Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition? $O(n)$



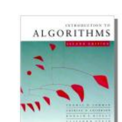
Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$



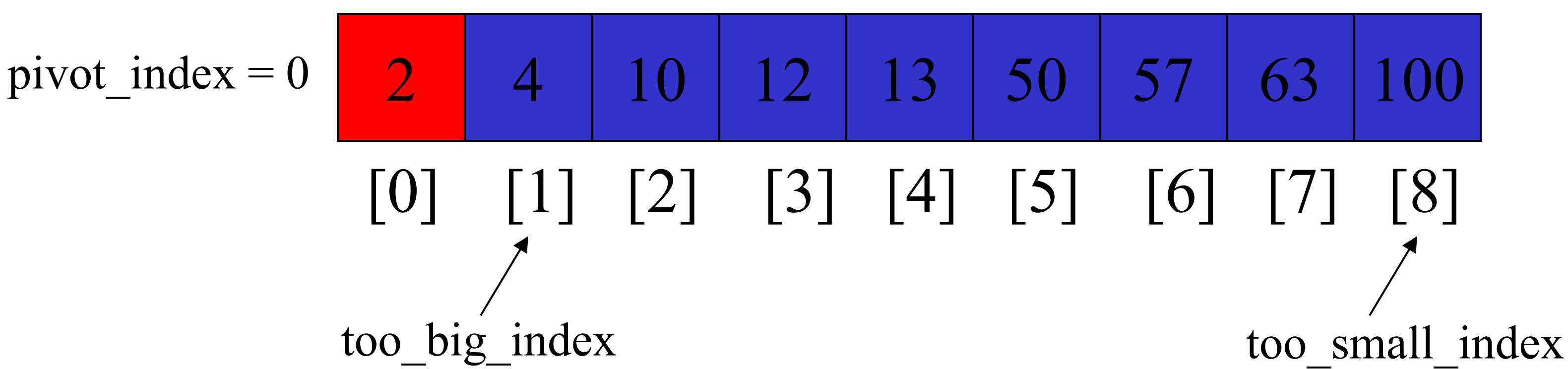
Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?



Quicksort: Worst Case

- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0	2	4	10	12	13	50	57	63	100
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
2	4	10	12	13	50	57	63	100

too_big_index

too_small_index

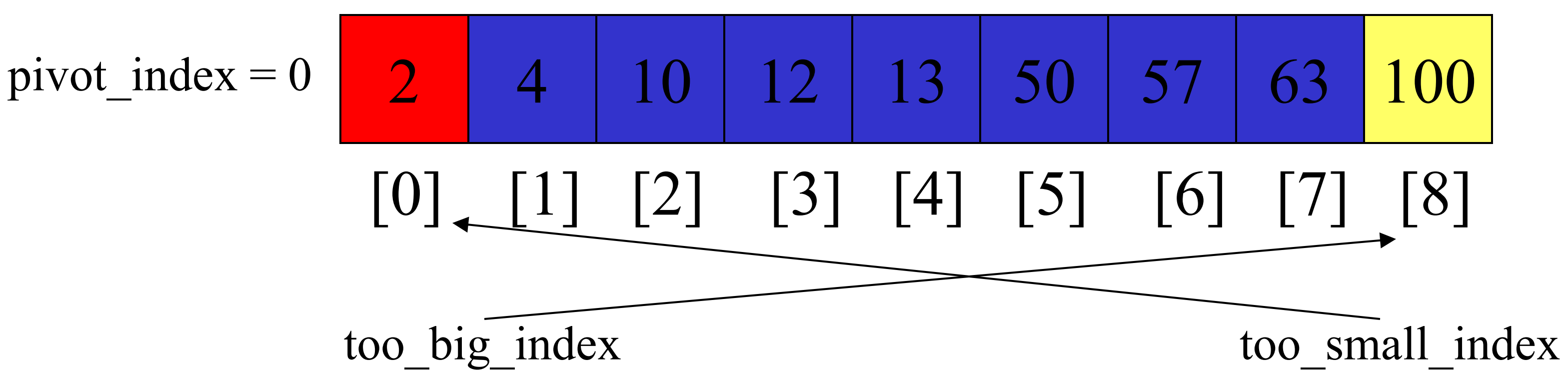
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

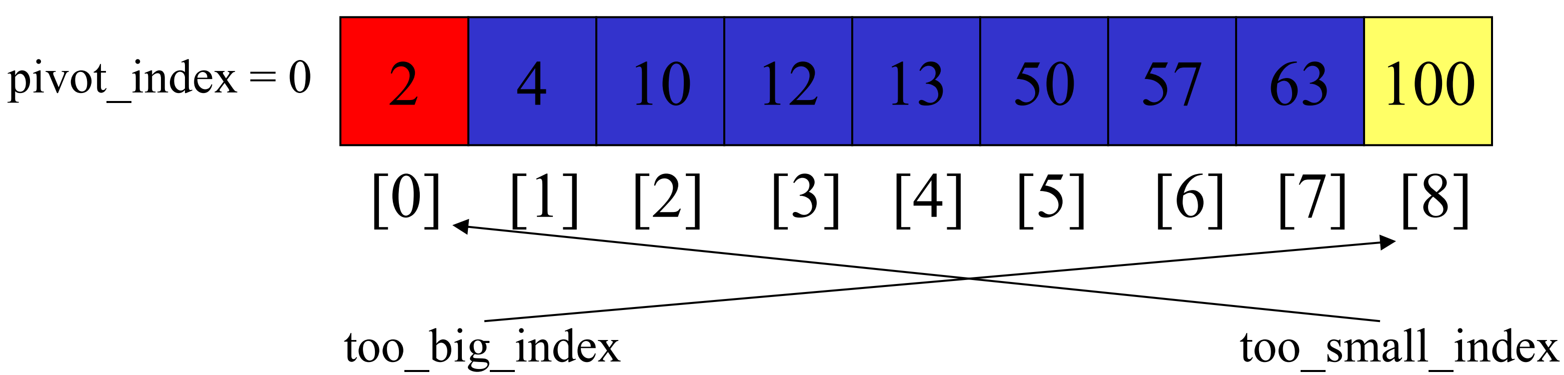
2	4	10	12	13	50	57	63	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

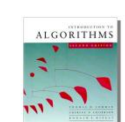
too_big_index too_small_index

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

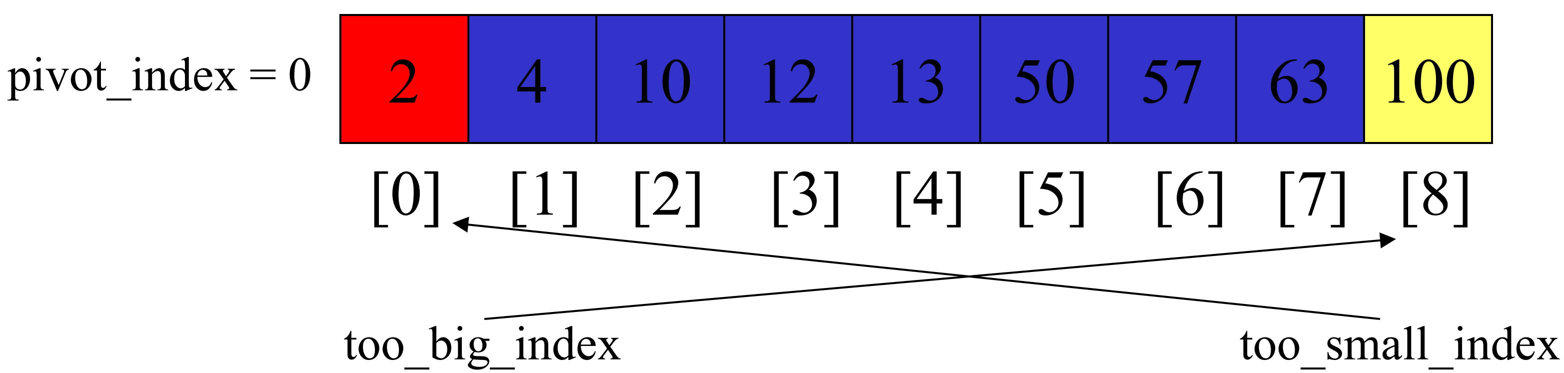


1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]





1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



1. While data[too_big_index] <= data[pivot]
    ++too_big_index
2. While data[too_small_index] > data[pivot]
    --too_small_index
3. If too_big_index < too_small_index
    swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

pivot_index = 0

2	4	10	12	13	50	57	63	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

<= data[pivot]   > data[pivot]

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!

Quicksort (A, r, s)

QuickSort (A, r, s)

if ($r \geq s$) return;
m = Partition (A, r, s);
$A1$ = QuickSort ($A, r, m\text{-}1$);
$A2$ = QuickSort ($A, m+1, s$);

In-place

Time Complexity :
$T(n) = T(m\text{-}1) + T(n\text{-}m) + O(n)$

Complexity

- $T(n) = T(m-1) + T(n-m) + n$

- Worst case:
  - $T(n) = T(0) + T(n-1) + n$
    $= T(n-1) + n$

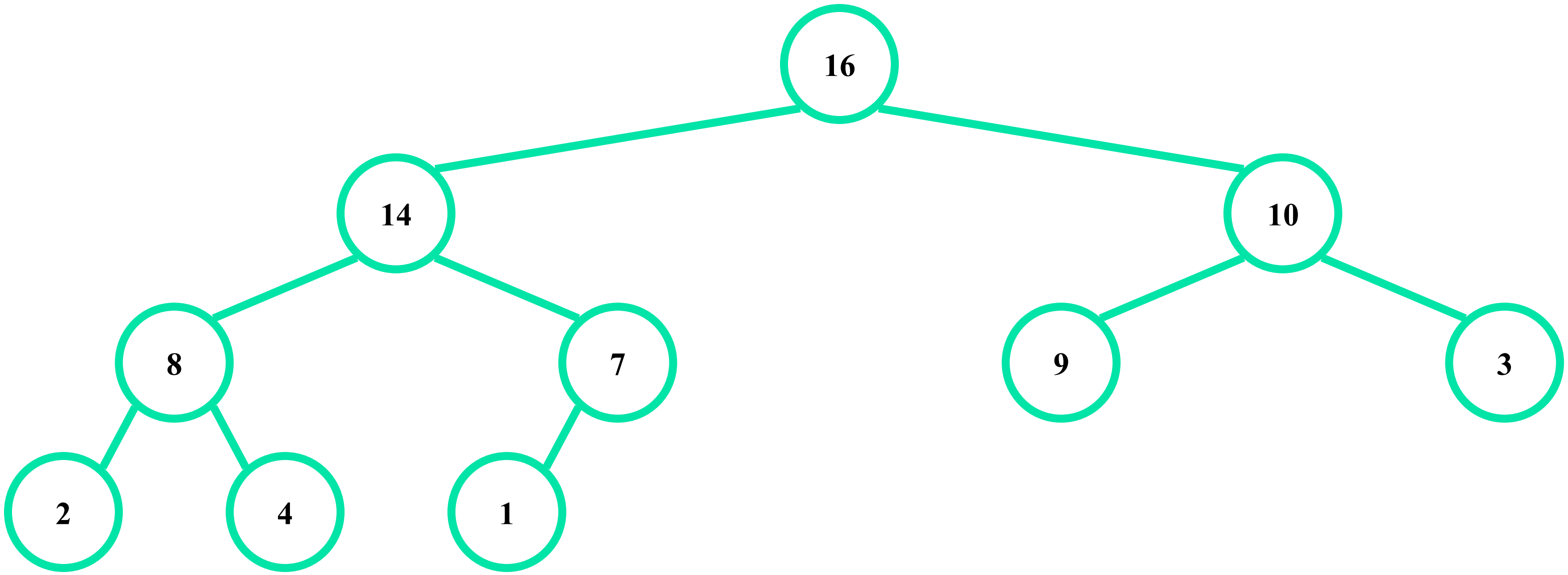
- Best case:
  - $T(n) = 2T(n/2) + n$

Sorting Revisited

- So far we've talked about three algorithms to sort an array of numbers
  - What is the advantage of merge sort?
  - What is the advantage of insertion sort?
  - What is the advantage of quick sort ?
- Next on the agenda: *Heapsort*
  - Combines advantages of previous algorithms

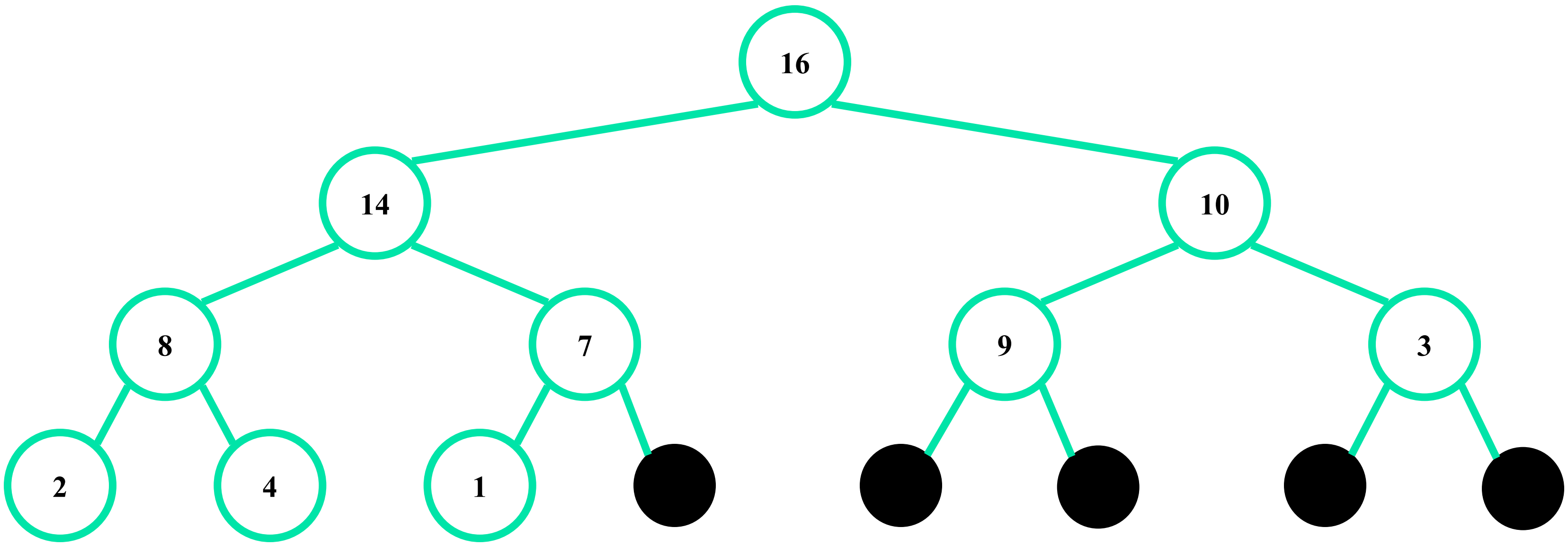
Heaps

- A *heap* can be seen as a complete binary tree:

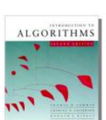


Heaps

- A *heap* can be seen as a complete binary tree:



  - The book calls them "nearly complete" binary trees; can think of unfilled slots as null pointers



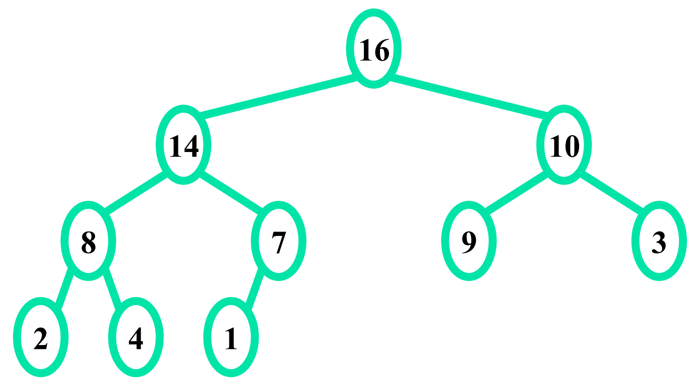
Heaps

- In practice, heaps are usually implemented as arrays:

A =

16	14	10	8	7	9	3	2	4	1

=



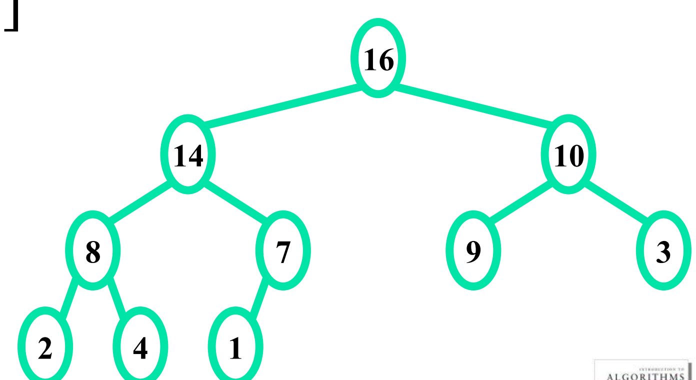
Heaps

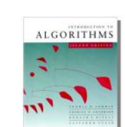
- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node *i* is A[*i*]
  - The parent of node *i* is A[*i*/2] (note: integer divide)
  - The left child of node *i* is A[2*i*]
  - The right child of node *i* is A[2*i* + 1]

A =

16	14	10	8	7	9	3	2	4	1

=





Referencing Heap Elements

- So…

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

- An aside: *How would you implement this most efficiently?*

The Heap Property

- Heaps also satisfy the *heap property*:

  $A[\mathit{Parent}(i)] \geq A[i]$ for all nodes $i > 1$

  - In other words, the value of a node is at most the value of its parent
  - *Where is the largest element in a heap stored?*

- Definitions:
  - The *height* of a node in the tree = the number of edges on the longest downward path to a leaf
  - The height of a tree = the height of its root

Heap Height

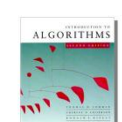
- *What is the height of an n-element heap?*
- basic heap operations take at most time proportional to the height of the heap

Heap Operations: Heapify()

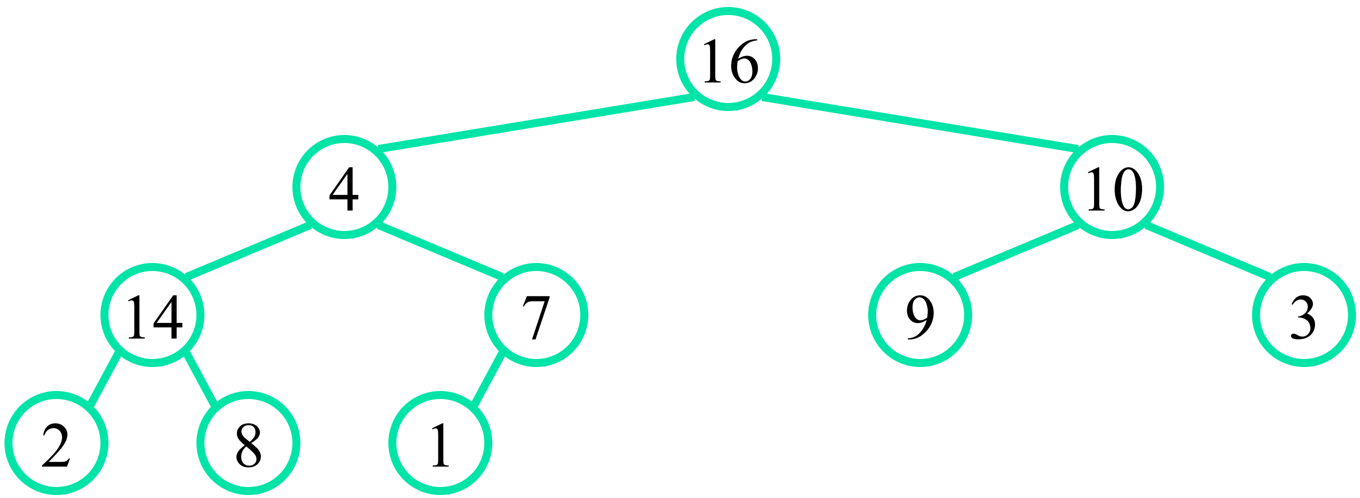
- **`Heapify()`**: maintain the heap property
  - Given: a node *i* in the heap with children *l* and *r*
  - Given: two subtrees rooted at *l* and *r*, assumed to be heaps
  - Problem: The subtree rooted at *i* may violate the heap property (*How?*)
  - Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property
    - *What do you suppose will be the basic operation between i, l, and r?*

Heap Operations: Heapify()

```
Heapify(A, i)
{
   l = Left(i); r = Right(i);
   if (l <= heap_size(A) && A[l] > A[i])
        largest = l;
   else
        largest = i;
   if (r <= heap_size(A) && A[r] > A[largest])
        largest = r;
   if (largest != i)
        Swap(A, i, largest);
        Heapify(A, largest);
}
```



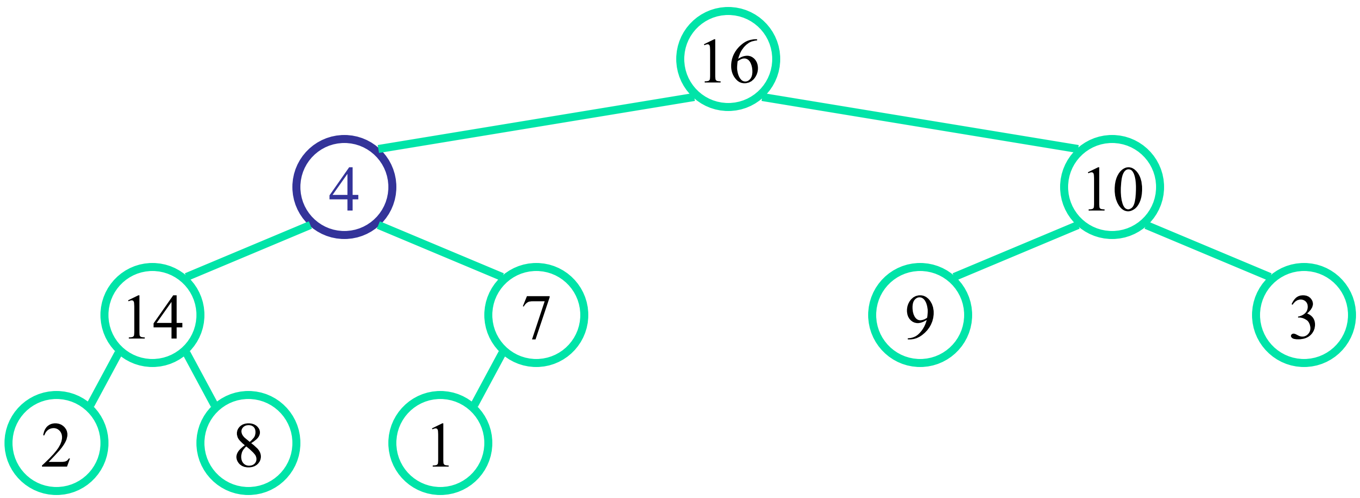
Heapify() Example



A =

16	4	10	14	7	9	3	2	8	1

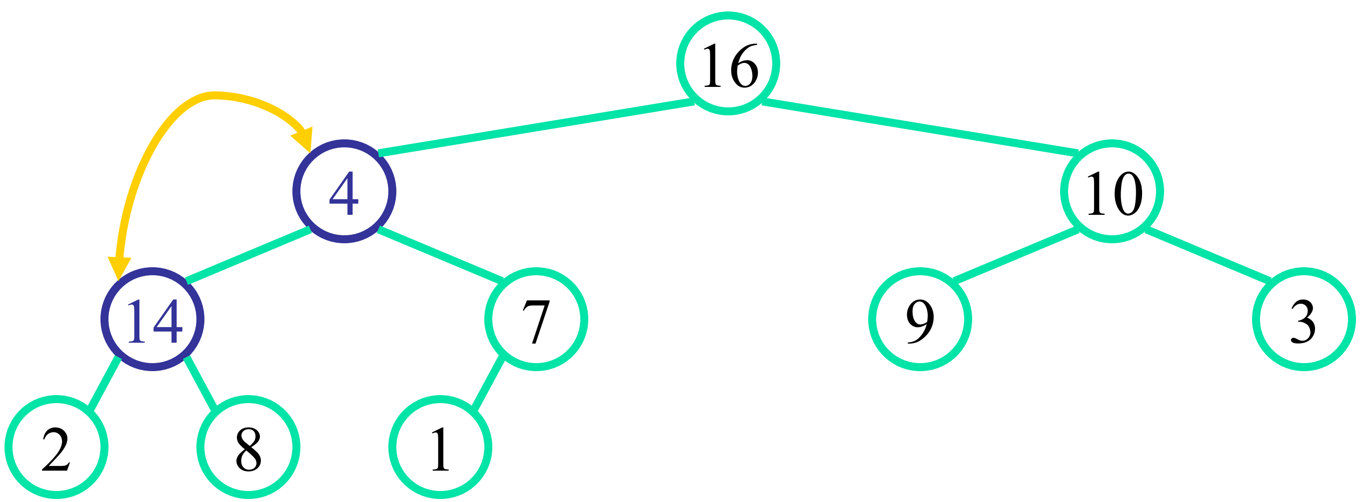
Heapify() Example



A =

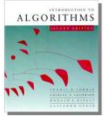
16	4	10	14	7	9	3	2	8	1

Heapify() Example

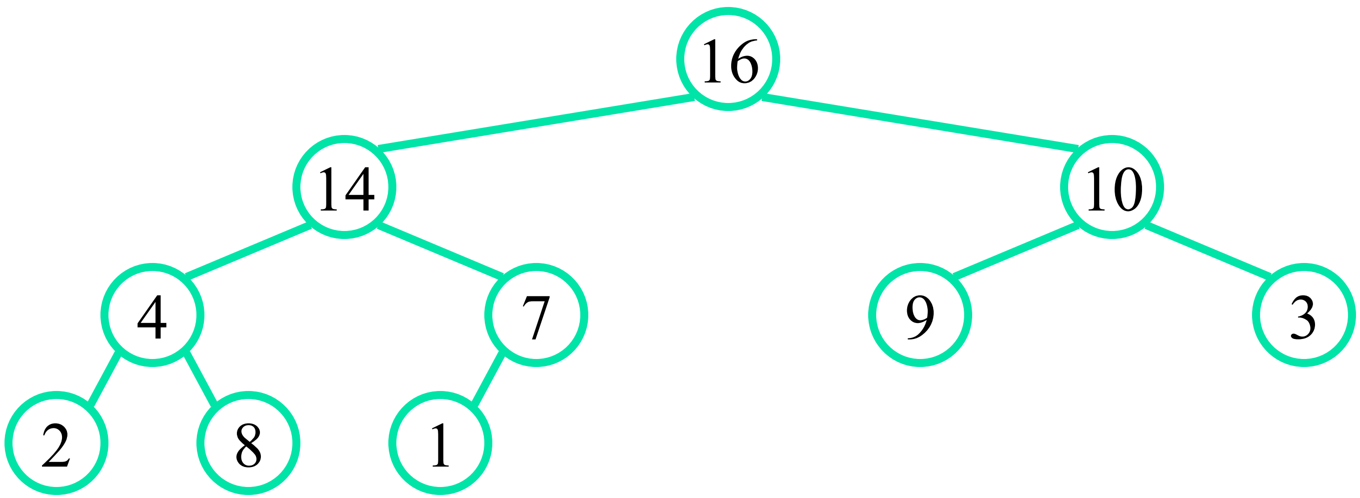


A =

16	4	10	14	7	9	3	2	8	1



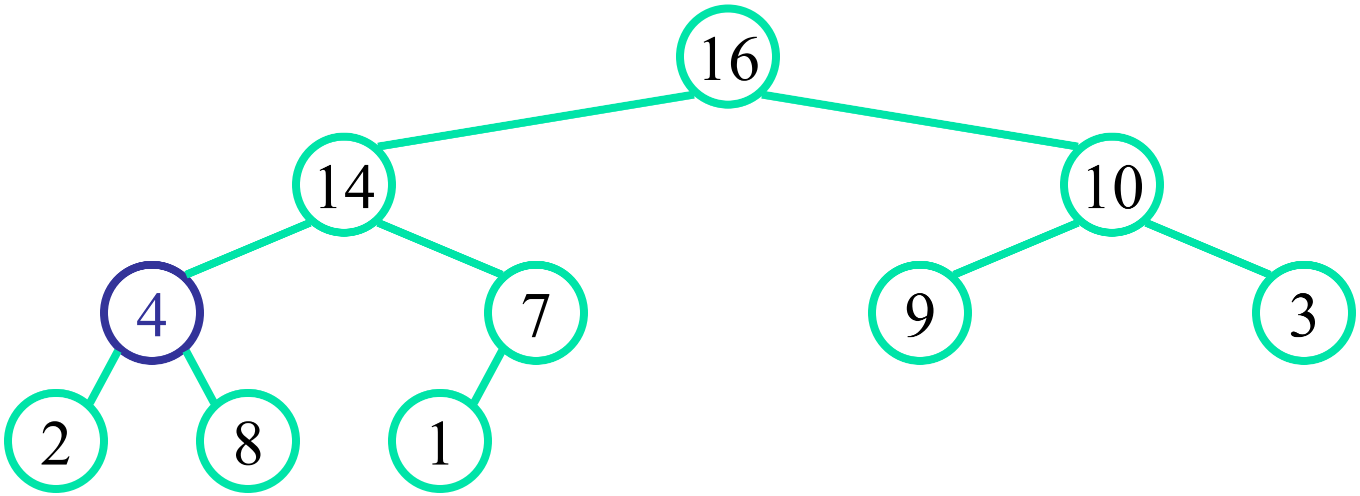
Heapify() Example



A =

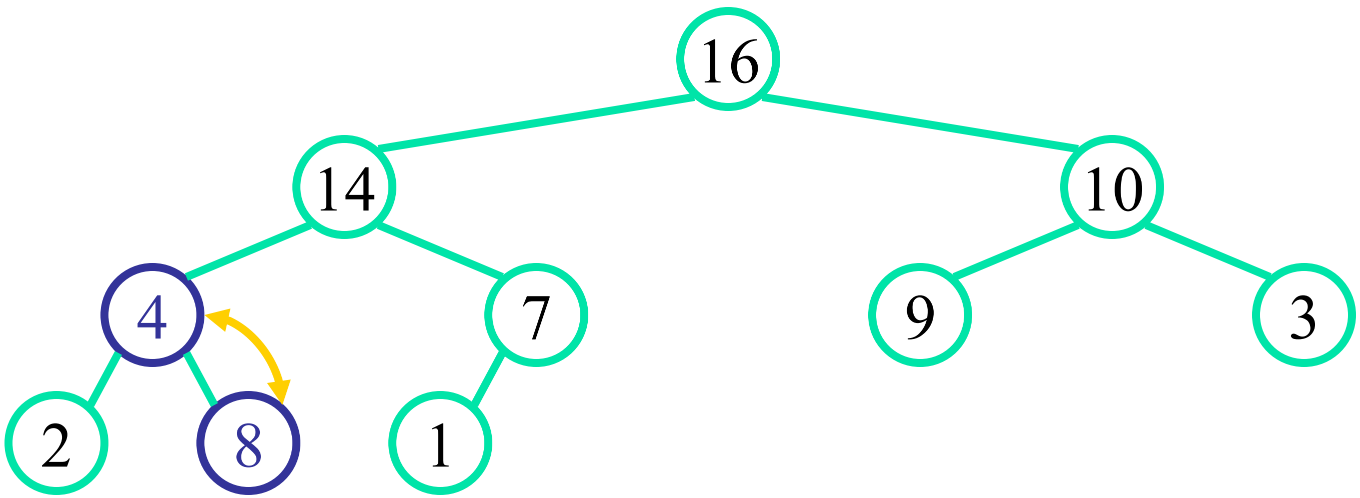
16	14	10	4	7	9	3	2	8	1

Heapify() Example



A =	16	14	10	4	7	9	3	2	8	1

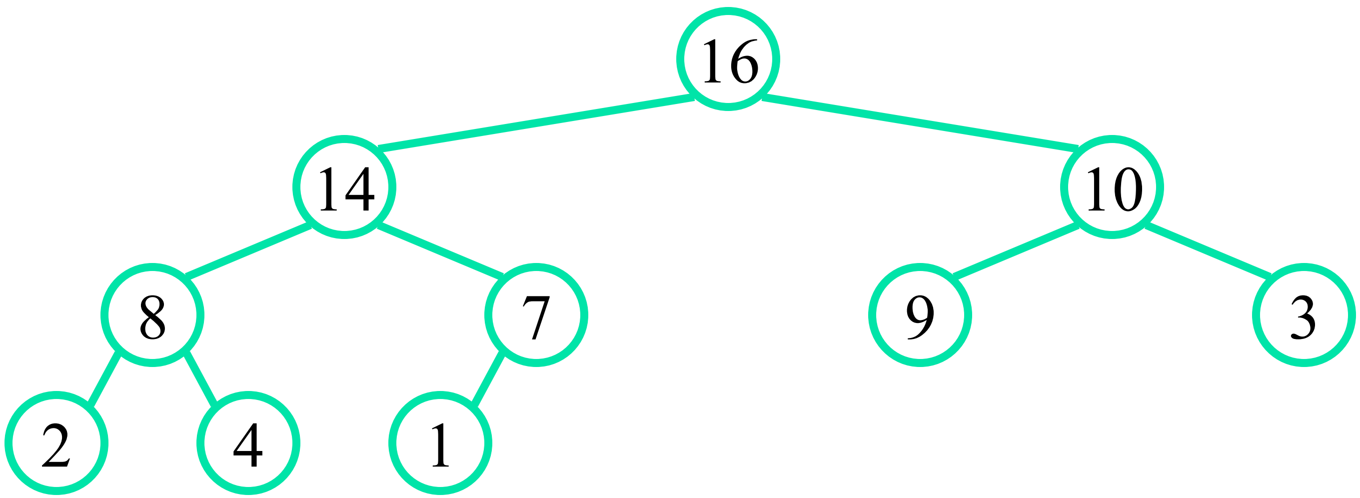
Heapify() Example



A =

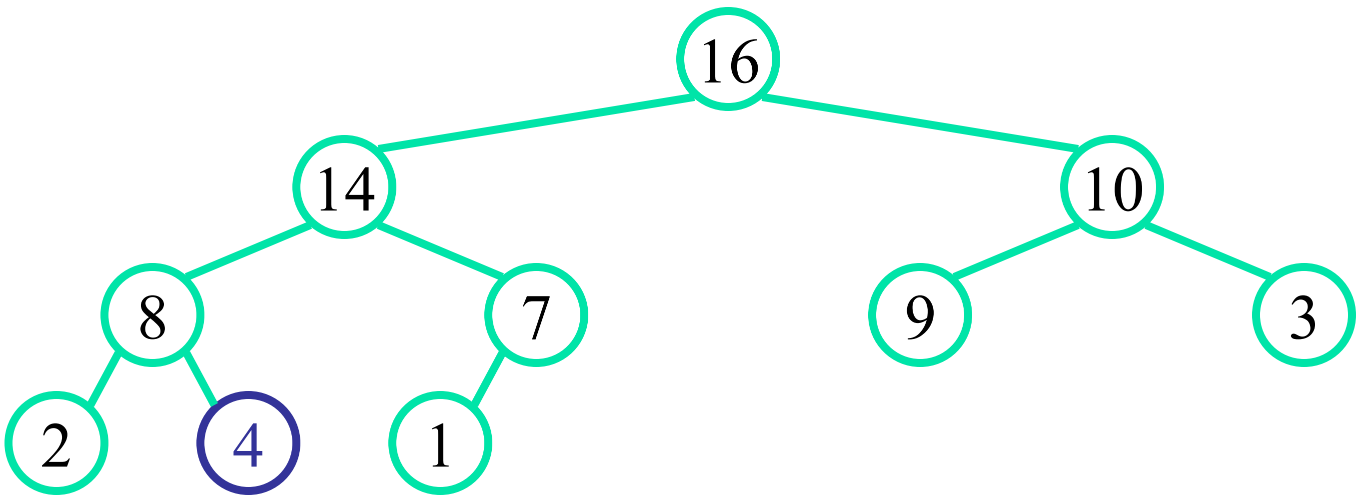
16	14	10	4	7	9	3	2	8	1

Heapify() Example



A =	16	14	10	8	7	9	3	2	4	1

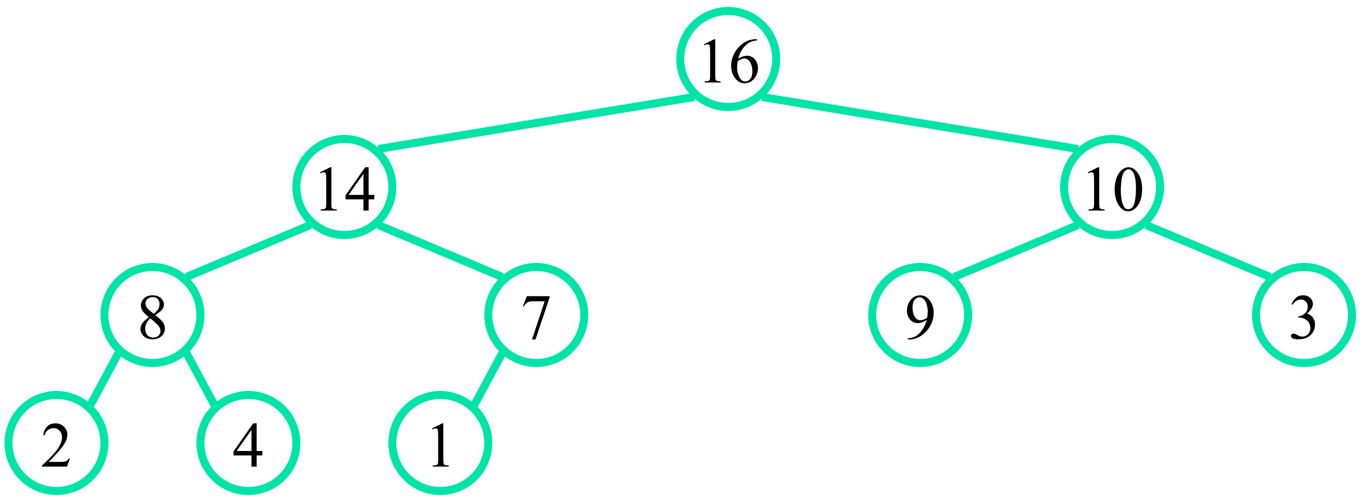
Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

Analyzing Heapify(): Formal

- Fixing up relationships between i, l, and r takes $\Theta(1)$ time
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*

  Answer: $2n/3$ (worst case: bottom row 1/2 full)
- So time taken by **`Heapify()`** is given by

  $T(n) \leq T(2n/3) + \Theta(1)$

Analyzing Heapify(): Formal

- So we have

  $T(n) \le T(2n/3) + \Theta(1)$
- By case 2 of the Master Theorem,

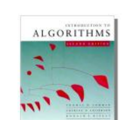
  $T(n) = O(\lg n)$
- Thus, **`Heapify()`** takes linear time

Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

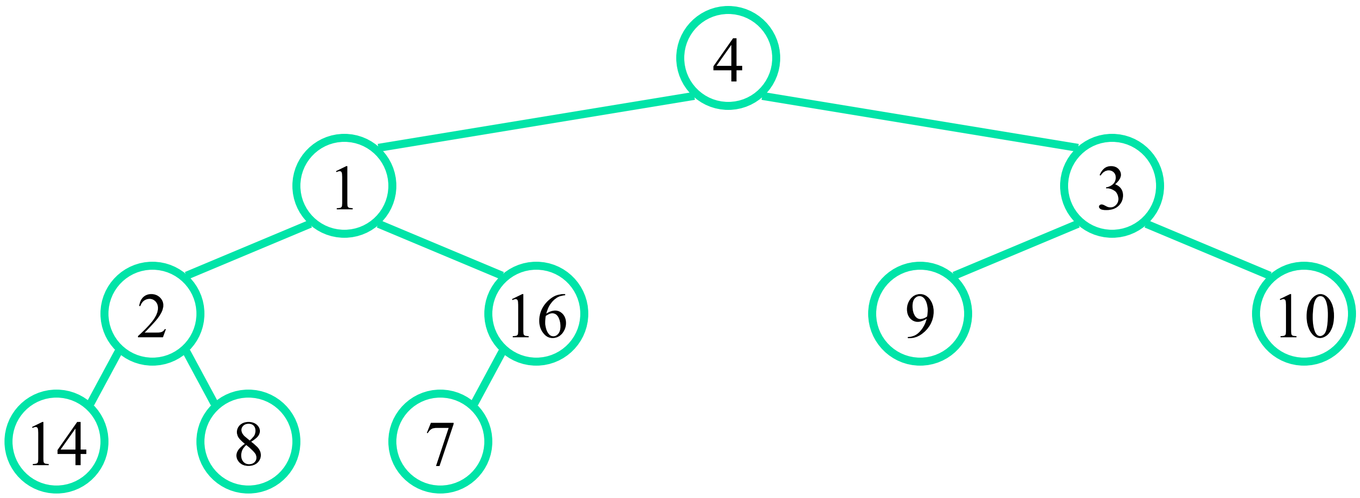
BuildHeap()

```
// given an unsorted array A, make A a
  heap
BuildHeap(A)
{
  heap_size(A) = length(A);
  for (i = ⌊length[A]/2⌋ downto 1)
      Heapify(A, i);
}
```



BuildHeap() Example

- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



Analyzing BuildHeap()

- Each call to **`Heapify()`** takes O(lg n) time
- There are O(n) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(n lg n)

Heapsort

- Given **`BuildHeap()`**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **`Heapify()`**
  - Repeat, always swapping A[1] for A[heap_size(A)]

Heapsort

```
Heapsort(A)
{
    BuildHeap(A);
    for (i = length(A) downto 2)
    {
        Swap(A[1], A[i]);
        heap_size(A) -= 1;
        Heapify(A, 1);
    }
}
```

Analyzing Heapsort

- The call to **`BuildHeap()`** takes O(n) time
- Each of the n - 1 calls to **`Heapify()`** takes O(lg n) time
- Thus the total time taken by **`HeapSort()`**
  $= O(n) + (n - 1)\, O(\lg n)$
  $= O(n) + O(n \lg n)$
  $= O(n \lg n)$